Programação Paralela e Distribuída

com MPI, OpenMP e OpenACC
para computação de alto desempenho

GABRIEL P. SILVA
CALEBE P. BIANCHINI
EVALDO B. COSTA

Casa do Código | alura

Programação Paralela e Distribuída

com MPI, OpenMP e OpenACC

Gabriel P. Silva

Calebe P. Bianchini

Evaldo B. Costa

ISBN

Impresso: 978-85-5519-303-3

Digital: 978-85-5519-304-0

Caso você deseje submeter alguma errata ou sugestão, acesse https://erratas.casadocodigo.com.br.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Silva, Gabriel P.
Programação paralela e distribuída : com MPI, OpenMP e OpenACC para computação de alto desempenho / Gabriel P. Silva, Calebe P. Bianchini, Evaldo B. Costa. -- São Paulo : Aovs Sistemas de Informática, 2022.

Bibliografia.
ISBN 978-85-5519-303-3

1. Algoritmos de computadores 2. Ciência da computação 3. Dados - Estruturas (Ciência da computação) - Estudo e ensino 4. Programação (Computadores) I. Bianchini, Calebe P. II. Costa, Evaldo B. III. Título.

22-109147 CDD-005.1

Índices para catálogo sistemático:

1. Programação de computadores 005.1

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Programação Paralela e Distribuída

Prefácio
Agradecimentos
1 Introdução
  1.1 Exemplos de aplicações paralelas
    1.1.1 Simulação oceânica
    1.1.2 Evolução das galáxias
    1.1.3 Petróleo e gás
  1.2 MPI
    1.2.1 Histórico
    1.2.2 Objetivos
  1.3 OpenMP
    1.3.1 Histórico
    1.3.2 Objetivos
  1.4 OpenACC
    1.4.1 Histórico
    1.4.2 Objetivos
2 Conceitos básicos
  2.1 Processos e Threads
    2.1.1 Processos
    2.1.2 *Threads*
    2.1.3 *Threads* de usuário e de núcleo
  2.2 Programação paralela
    2.2.1 Programação com Memória compartilhada
    2.2.2 Programação por Troca de mensagens
    2.2.3 Sincronização de barreira
    2.2.4 Particionamento de dados e tarefas
  2.3 Balanceamento de carga
    2.3.1 Balanceamento estático
    2.3.2 Balanceamento dinâmico

2.4 Avaliação de desempenho
2.4.1 Medidas de desempenho
2.4.2 Lei de Amdahl
2.4.3 Lei de Gustafson
2.5 Arquiteturas paralelas
2.5.1 Arquiteturas SISD
2.5.2 Arquiteturas SIMD
2.5.3 Arquiteturas MIMD
2.5.4 Aceleradores
2.6 Exercícios propostos
3 Comunicação ponto a ponto no MPI
3.1 Introdução
3.2 Comunicadores
3.3 Exemplo de um programa em MPI
3.4 Funções de gerenciamento do ambiente
3.4.1 Iniciando e finalizando o MPI
3.4.2 Medindo o tempo de execução
3.4.3 Abortando um programa
3.4.4 Verificando a versão da biblioteca MPI
3.4.5 Verificando o nome do processador
3.5 Envio e recepção de mensagens
3.5.1 Enviando uma mensagem
3.5.2 Recebendo uma mensagem
3.5.3 Correspondência entre os tipos MPI e C
3.6 Identificando as mensagens recebidas
3.6.1 Utilizando as informações em MPI_Status
3.6.2 Verificando o tamanho das mensagens recebidas
3.7 Algumas recomendações
Erros frequentes
Otimizando o desempenho
3.8 Estudo de caso: método do trapézio
3.8.1 Avaliação de desempenho
3.9 Exercícios propostos
4 Comunicação coletiva no MPI
4.1 Barreira
4.2 Difusão

4.3 Distribuição
4.4 Coleta
4.5 Redução
Operadores de redução MPI_MAXLOC e MPI_MINLOC
4.6 Redução com difusão
4.7 Coleta com difusão
4.8 Transposição
4.9 Algumas observações
Recomendações
Otimizando o desempenho
4.10 Estudo de caso: multiplicação de matriz por vetor
Avaliação de desempenho
4.11 Exercícios propostos
5 Comunicação MPI em detalhes
5.1 Introdução
5.2 Rotinas de envio e recepção bloqueantes
5.3 Rotinas de envio e recepção não bloqueantes
5.4 Esperando a mensagem
5.5 Modos de comunicação
5.5.1 Modo síncrono
5.5.2 Modo pronto
5.5.3 Modo bufferizado
5.5.4 Modo padrão
5.6 Evitando o impasse ou deadlock
5.7 Considerações de desempenho
5.8 Estudo de caso: números primos
Avaliação de desempenho
5.9 Exercícios propostos
6 OpenMP
6.1 Introdução
6.2 Diretivas principais
6.2.1 Diretiva parallel
6.2.2 Diretiva single
6.2.3 Diretiva sections
6.2.4 Diretiva parallel sections
6.2.5 Diretiva for

6.2.6 Diretiva parallel for
6.3 Funções OPENMP
6.3.1 Número de threads na região paralela
6.3.2 Medindo o tempo de execução
6.4 Cláusulas
6.4.1 Cláusulas da diretiva parallel
6.4.2 Cláusulas da diretiva single
6.4.3 Cláusulas da diretiva for
6.4.4 Cláusulas da diretiva sections
6.5 Sincronização
6.5.1 Diretiva barrier
6.5.2 Diretiva master
6.5.3 Cláusula nowait
6.5.4 Diretiva critical
6.5.5 Diretiva atomic
6.5.6 Funções lock
6.6 Variáveis de ambiente
6.7 Erros comuns e recomendações
6.8 Estudos de caso
6.8.1 Estudo de caso: método do trapézio
6.8.2 Estudo de caso: contagem de números primos
6.8.3 Estudo de caso: método de Jacobi
6.9 Exercícios
7 OpenACC
7.1 Modelo de programação OpenACC
7.1.1 Níveis de paralelismo do OpenACC
7.2 Diretivas principais
7.2.1 Diretiva kernels
7.2.2 Diretiva parallel
7.2.3 Diretiva kernels vs diretiva parallel
7.2.4 Diretivas parallel loop e kernels loop
7.3 Movimentação de dados
7.3.1 Diretiva data
7.3.2 Memória Unificada (Unified Memory)
7.4 Cláusulas das diretivas parallel
7.4.1 Cláusulas private e firstprivate

7.4.2 Cláusula default(none)
7.4.3 Cláusula independent
7.4.4 Cláusula reduction
7.4.5 Cláusula if
7.4.6 Cláusulas num_gangs, num_workers e vector_length
7.5 Cláusulas da diretiva loop
7.5.1 Cláusula gang
7.5.2 Cláusula worker
7.5.3 Cláusula vector
7.5.4 Cláusula seq
7.5.5 Cláusula auto
7.5.6 Cláusula private
7.5.7 Cláusula reduction
7.5.8 Cláusula collapse
7.5.9 Cláusula tile
7.5.10 Cláusula device_type
7.6 Diretivas avançadas
7.6.1 Diretiva routine
7.6.2 Diretiva atomic
7.7 Funções OpenACC
7.8 Variáveis de ambiente
7.9 Erros comuns e recomendações
7.10 Estudos de caso
7.10.1 Estudo de caso: cálculo de Pi
7.10.2 Estudo de caso: método do trapézio
7.10.3 Estudo de caso: método Jacobi
7.11 Exercícios
8 Ambientes de execução
8.1 Preparação do ambiente de execução MPI
8.1.1 Introdução
8.1.2 Instalação
8.1.3 Compilação
8.1.4 Configurando o SSH
8.2 Preparação do Ambiente de Execução OpenMP
8.2.1 Compiladores
8.2.2 Variáveis de ambiente

8.3 Preparação do ambiente de execução OpenACC
8.3.1 Introdução
8.3.2 Compiladores
8.3.3 Variáveis de ambiente
Referências

Prefácio

> "Dar-me-eis um grão de trigo pela primeira casa do tabuleiro; dois pela segunda, quatro pela terceira, oito pela quarta, e, assim dobrando sucessivamente, até a sexagésima quarta e última casa do tabuleiro. Peço-vos, ó rei, de acordo com a vossa magnânima oferta, que autorizeis o pagamento em grãos de trigo, e assim como indiquei!"
>
> — Malba Tahan, *O Homem que Calculava*

A pesquisa científica moderna, nas diversas áreas de conhecimento, tem desenvolvido modelos computacionais sofisticados para a solução de problemas cada vez mais complexos. Esses modelos são transformados em aplicações paralelas que realizam simulações com o intuito de obter, no menor prazo possível, aproximações cada vez mais precisas da realidade.

Essas aplicações paralelas são construídas com uso de interfaces de programação e bibliotecas associadas a linguagens de programação convencionais tais como C e FORTRAN, fazendo o uso de plataformas de computação paralelas como *clusters*, sistemas multiprocessadores de memória compartilhada e aceleradores com alto poder computacional para alcançar o desempenho desejado.

A formação de recursos humanos especializados nesta área demanda um longo tempo de investimento e deve ser iniciada o mais cedo possível, já nos primeiros anos dos cursos de graduação, seja nos cursos de Engenharia da Computação, Ciência da Computação ou Engenharia de Software.

Este livro tem como objetivo apresentar conceitos iniciais de programação paralela para estudantes de graduação, pós-graduação e profissionais da área. Fazemos uso da linguagem C para

apresentar exemplos simples e práticos com uso das interfaces de programação e bibliotecas MPI, no paradigma de troca de mensagens, além de OpenMP, no paradigma de memória compartilhada e OpenACC, para uso com aceleradores. Longe de ser um compêndio de referência, as facilidades que esses padrões oferecem para obtenção de um código paralelo simples e eficiente são apresentadas gradativamente, junto de uma explanação sobre como criar programas paralelos básicos.

O MPI é uma das interfaces de programação paralela mais utilizadas na computação científica, podendo ser empregada desde os equipamentos mais simples, com apenas algumas dezenas de processadores, até os *clusters* de alto desempenho, com dezenas de milhares de processadores. O OpenMP e o OpenACC, por suas vezes, são padrões para programação de sistemas multiprocessadores com memória compartilhada e aceleradores como GPUs e processadores *manycores*, caracterizados pela sua extrema facilidade de uso e relativo baixo custo.

Este livro está organizado da seguinte maneira: primeiramente são apresentados conceitos gerais de programação paralela, os diversos paradigmas de programação e formas de desenvolvimento de um programa paralelo, com considerações sobre balanceamento de carga e as métricas de avaliação de desempenho, comuns a qualquer tipo de programa paralelo.

No capítulo seguinte as funções básicas para o envio e recepção de mensagens do MPI são introduzidas. Logo após, as funções de comunicação coletiva, de grande importância para o trabalho cooperativo entre processos, são descritas. Em seguida, são apresentados detalhes sobre os diversos modos disponíveis no MPI para o envio e recepção de mensagens. No capítulo sobre o OpenMP, as diversas diretivas e suas respectivas cláusulas que são utilizadas para explorar o paralelismo embutido nos laços computacionais são apresentadas, junto das primitivas de

sincronização disponíveis na linguagem, essenciais no paradigma de memória compartilhada.

No último capítulo, as diretivas e cláusulas do OpenACC são discutidos. O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC. O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore, manycore* e GPUs.

O livro inclui também apêndices com detalhes dos ambientes de execução do MPI, OpenMP e OpenACC. São discutidas as diferenças entre os diversos pacotes e compiladores disponíveis, além das opções de compilação e outros detalhes para extrair o máximo desempenho das aplicações.

Ao longo deste livro são apresentados exemplos simples e objetivos para o uso de cada uma das funções, diretivas e cláusulas dos diversos paradigmas, bibliotecas e interfaces de programação abordados. Todos os exemplos foram cuidadosamente elaborados, compilados e testados em ambientes paralelos, de modo que possam ser baixados, compilados e reproduzidos facilmente em qualquer equipamento onde um mínimo de paralelismo esteja disponível. Estudos de caso e exercícios propostos podem ser encontrados ao final de cada capítulo, como forma de fixação e complemento dos conceitos elencados na parte teórica.

Esperamos que este livro possa ser um guia seguro para os passos iniciais das pessoas interessadas no uso da programação paralela de uma forma eficiente e produtiva.

Rio de Janeiro, 15 de março de 2022
Gabriel P. Silva

Calebe P. Bianchini
Evaldo B. Costa

Agradecimentos

Gostaríamos de agradecer pelos comentários construtivos de diversos colegas, dos alunos e, em especial, à professora Silvana Rossetto, pela inestimável colaboração na elaboração do segundo e terceiro capítulo desta obra. À professora Denise Stringhini pela sua paciente revisão e comentários ao texto.

1 Introdução

Neste capítulo são apresentados exemplos de aplicações reais cuja solução pode ser acelerada com o uso da programação paralela, além de um breve histórico e objetivos pretendidos com a criação dos padrões MPI, OpenMP e OpenACC.

1.1 Exemplos de aplicações paralelas

Quais são os tipos de aplicação beneficiadas com o uso da programação paralela? Embora a programação paralela esteja presente em diversos tipos de aplicações, incluindo aquelas no seu aparelho celular e no computador pessoal, certamente apresenta as maiores vantagens de uso nas aplicações científicas. Por serem aplicações que realizam uma quantidade expressiva de cálculos sobre um conjunto significativo de dados, o uso da programação paralela permite a obtenção de resultados em prazos muito mais reduzidos, quando comparado com o uso da programação sequencial.

A previsão do tempo, por exemplo, deve ser feita antes que uma enchente, tornado ou furacão atinja uma determinada região. Quanto mais precisa e antecipada for a informação, por exemplo, de quando e por onde um furacão vai passar, mais vidas poderão ser salvas e a defesa civil terá melhores condições para se organizar para esse evento climático extremo.

Além desse, podemos observar uma lista na Tabela 1.1, extensa mas não exaustiva, apresentando exemplos de aplicações científicas, cujos ganhos são significativos com o uso da programação paralela:

Tabela 1.1: Exemplos de aplicações paralelas

Genoma Humano	**Turbulência dos Fluidos**
Dinâmica de Veículos	Dinâmica de Fluidos Viscosos
Modelagem de Supercondutores	Cromodinâmica Quântica
Visão por Computador	Farmacêutica
Biologia Estrutural	Previsão do Tempo
Álgebra Linear Densa	Métodos Espectrais
ou Esparsa	(Transformada de Fourier)
N-Body (Barnes-Hut)	Programação Dinâmica
Grade Não Estruturada	Simulação de
(Análise de Elementos Finitos)	Monte Carlo
Lógica Combinatória (Criptografia)	Grafos (Ordenação)
Grade Estruturada	Modelos Gráficos para Raciocínio
(Lattice Boltzman)	(Redes Bayesianas)
Métodos Branch	Simulação de Máquinas
and Bound	de Estado Finito

Algumas dessas aplicações são destacadas e comentadas a seguir.

1.1.1 Simulação oceânica

O modelo do clima da Terra precisa saber como a atmosfera interage com os oceanos, os quais ocupam 3/4 da superfície da Terra. Estão envolvidos nesse estudo diversas forças físicas: efeitos atmosféricos, vento e fricção com o fundo dos oceanos. Para esse processo de modelagem, o oceano é dividido em planos e cada plano possui uma grade de pontos igualmente espaçados (uma matriz) com informações como temperatura, salinidade, velocidade e pressão, entre outras (Marshall et al. 1997). Além disso, todos esses planos são simulados para diversos intervalos de tempo, também igualmente espaçados.

Para o caso do oceano Pacífico, por exemplo, cuja maior dimensão é de 17.700 km x 14.500 km, a simulação pode ser feita com uma grade de 1.520 x 1.088 pontos, resultando em uma distância de 12,5 km entre os pontos, com cerca de 30 camadas (planos) de profundidade. Para simular o comportamento do oceano Pacífico para um período de 5 anos, atualizados a cada 8 horas, serão necessários cerca de 5.500 intervalos de tempo, cada um deles com cerca de 50.000 pontos representando o oceano naquele instante.

Para esse tipo de aplicação, cujo resultado da simulação para todo oceano Pacífico é apresentado na Figura 1.1 (extraída de (Wang et al. 2005)), é possível realizar boa parte do seu processamento de forma paralela, pois há certa independência entre os diversos pontos da grade. A divisão do processamento em várias tarefas executadas em paralelo pelos diversos processadores (256 neste exemplo) permitiu uma redução significativa do tempo total de simulação.

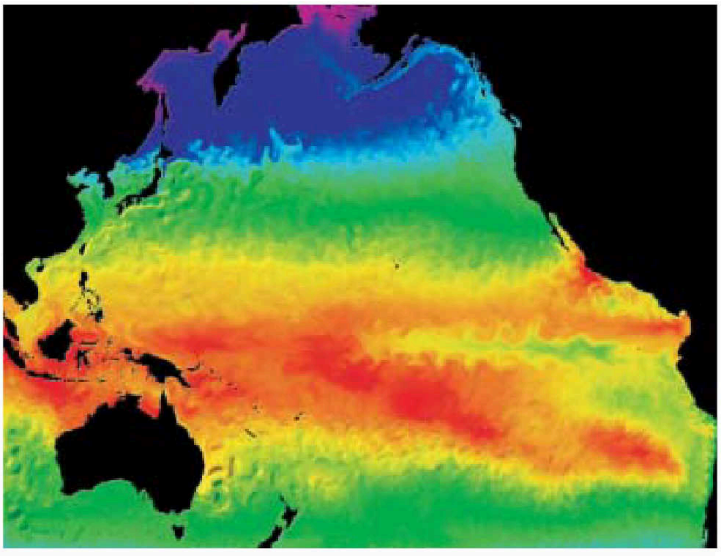

Figura 1.1: Simulação oceânica

1.1.2 Evolução das galáxias

A simulação da evolução de galáxias ao longo de milhões de anos é um problema muito interessante, cuja solução pode ser acelerada com uso da computação paralela. Usando simples equações de física

Newtoniana, como a massa e a distância de suas estrelas, é possível prever com precisão o comportamento futuro de um conjunto de galáxias. Contudo, considerando-se que uma galáxia possui milhões de estrelas, esta seria uma aplicação com custos computacionais elevadíssimos.

Para esse tipo de problema seria necessário calcular a interação gravitacional para cada par de estrelas em diversos intervalos de tempo. Esse método de solução tem complexidade O(Math input error), de aplicação impossível para os milhões de estrelas de uma galáxia. Mas levando-se em conta que a força gravitacional diminui com o quadrado da distância, podemos usar um algoritmo de complexidade O(Math input error).

Nesse caso, grupos de estrelas mais distantes são consideradas como uma única estrela com massa equivalente e situada no centro de massa dessas galáxias. Esse algoritmo hierárquico recebeu o nome de Barnes-Hut (Barnes and Hut 1986) e sua estrutura permite explorar uma ampla concorrência entre estrelas dentro de um intervalo de tempo. Mas, como são padrões bastantes irregulares e variantes no tempo, é um desafio explorar esta concorrência em arquiteturas paralelas e até hoje vários artigos tem sido publicados abordando esse tema.

A Figura 1.2, obtida a partir da referência (Berg 2016), apresenta o resultado da distribuição de 5.000 partículas, representando duas galáxias, entre os nós computacionais segundo algoritmo de Barnes-Hut.

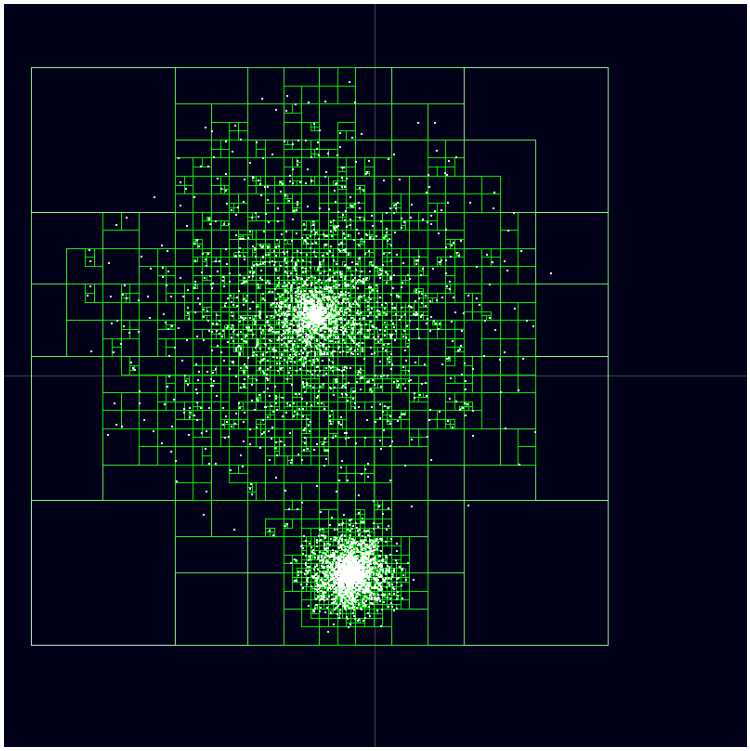

Figura 1.2: Simulação Barnes-Hut

1.1.3 Petróleo e gás

A exploração petrolífera e a prospecção mineral são as atividades, em nível de indústria de aproveitamento de recursos naturais, de maior aplicação da geofísica. A geofísica é uma ciência cujo objetivo é o estudo indireto da Terra, através de métodos baseados na observação física de campos e ondas. Em geral, o estudo geofísico é realizado em três fases:

- Aquisição
- Processamento
- Interpretação

Recentemente, o avanço tecnológico na área da computação tem produzido uma superposição cada vez maior entre essas três etapas. A prospecção mineral tem como área de interesse grandes corpos de

rochas cristalinas entrecortadas por um complexo padrão estrutural. A geofísica, por ser um método de investigação indireta, permite a definição de reservatórios petrolíferos ou gasíferos no interior das bacias sedimentares, bem como auxilia na definição de macroestruturas responsáveis pelo jazimento de corpos mineralizados.

Em particular, cabe enfatizar os métodos sísmicos que, pelo alto poder de resolução, têm sido largamente utilizados na exploração e desenvolvimento de reservatórios de petróleo e gás. Baseado na propagação de ondas sísmicas geradas por uma fonte artificial, e registradas em sensores denominados geofones (na terra) e hidrofones (na água) (veja a Figura 1.3, extraída de (Panetta et al. 2012)), o geofísico usa os métodos sísmicos na definição de imagens de rochas na sub-superfície da Terra, em duas (2D) e três (3D) dimensões, possibilitando a delimitação de alvos de interesse para a exploração.

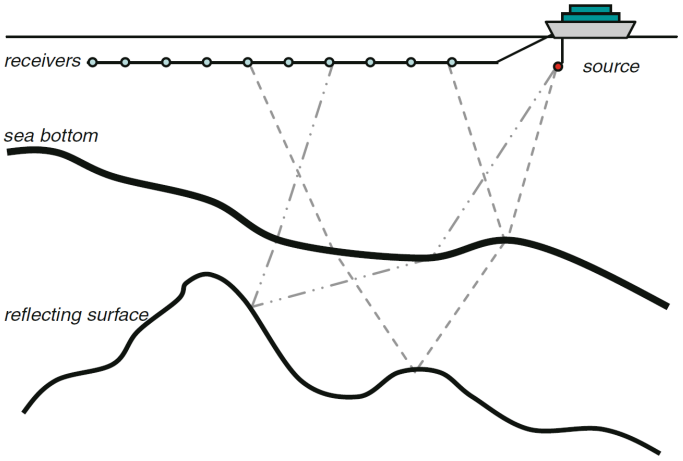

Figura 1.3: Aquisição de dados sísmicos

Essa tarefa exige sofisticados equipamentos para a aquisição de uma gigantesca quantidade de dados, e sistemas computacionais de alto desempenho para o processamento e interpretação 2D e 3D (Antonio Carrillo-Ledesma and Ismael Herrera 2013). Novamente, a computação paralela é uma alternativa para a realização desse tipo de processamento dentro de intervalos de tempo razoáveis.

1.2 MPI

1.2.1 Histórico

O MPI (*Message Passing Interface*) é um padrão de troca de mensagens portátil que facilita o desenvolvimento de aplicações paralelas, sendo utilizado tanto em ambientes de memória compartilhada como de memória distribuída. Há várias implementações do padrão MPI — na forma de bibliotecas de funções — disponíveis nas linguagens C, C++ e Fortran.

O projeto do MPI procurou utilizar os melhores recursos disponíveis em um grande número de sistemas de troca de mensagem existentes, ao invés de selecionar um deles e utilizá-lo como padrão. Logo, o MPI foi fortemente influenciado por trabalhos precursores realizados em: IBM T. J. Watson Research Center, Intel's NX2, Express, nCUBE's Vertex e PARMACS. Outras contribuições importantes vieram do Zipcode, Chimp, PVM, Chameleon e PICL (Marc et al. 1996).

As principais vantagens da criação de um padrão de troca de mensagens para uso em ambientes de memória distribuída são a portabilidade e a facilidade de uso. O padrão fornece aos fabricantes uma definição clara de um conjunto básico de rotinas, possibilitando implementações eficientes, em alguns casos com suporte específico de *hardware*, e favorecendo a escalabilidade.

O esforço de padronização do MPI envolveu dezenas de professores e pesquisadores de diversas organizações dos Estados Unidos e Europa. A maioria dos fabricantes de computadores paralelos estiveram envolvidos nesse esforço, junto a pesquisadores das universidades, laboratórios do governo e empresas. O processo de padronização começou com o "Workshop on Standards for Message Passing in a Distributed Memory Environment", patrocinado pelo Center for Research on Parallel Computing, realizado em abril de

1992 (Walker 1992). Estabeleceu-se então um grupo de trabalho para a elaboração da primeira versão do padrão MPI. Essa versão foi apresentada na conferência "Supercomputing", realizada em novembro de 1993 e ficou conhecida como a versão 1.0 do MPI.

Posteriormente, surgiram versões de correções e esclarecimentos da versão 1.0, nominadamente, MPI-1.1 (Marc et al. 1996), MPI-1.2 e MPI-1.3. O documento com o padrão final do MPI-2.1, com extensões para operações de E/S em paralelo e gerenciamento dinâmico de processos, foi finalizado em 2008. A versão 3.0, que apresenta uma atualização expressiva do padrão MPI, foi aprovada em 2012. As atualizações incluem extensões para operações coletivas com versões não bloqueantes, entre outras. A versão 3.1 veio em seguida, com a habilidade de utilizar o paradigma de memória compartilhada dentro de um nó, no ano de 2015 (MPI Forum 2015).

- MPI-1 versão final em 1994;
    - foi seguido pelas versões MPI-1.1 (Junho 1995), MPI-1.2 (Julho 1997) e MPI-1.3 (Maio 2008).
- MPI-2 versão final em 1996;
    - foi seguido pelas versões MPI-2.1 (Setembro 2008) e MPI-2.2 (Setembro 2009).
- MPI-3.0 versão final em 2012;
    - foi seguido pela versão MPI-3.1 (Junho de 2015).

Podemos ver um resumo dessa evolução ilustrada na Figura 1.4 [1]. Assim sendo, um programa poderá ter um dos seguintes níveis de compatibilidade com o padrão MPI:

- Compatibilidade com o MPI-1 significa compatibilidade com o MPI-1.3.

- Compatibilidade com o MPI-2 significa compatibilidade com a especificação MPI-2.2.
- Compatibilidade com o MPI-3 significa compatibilidade com a especificação MPI-3.1.

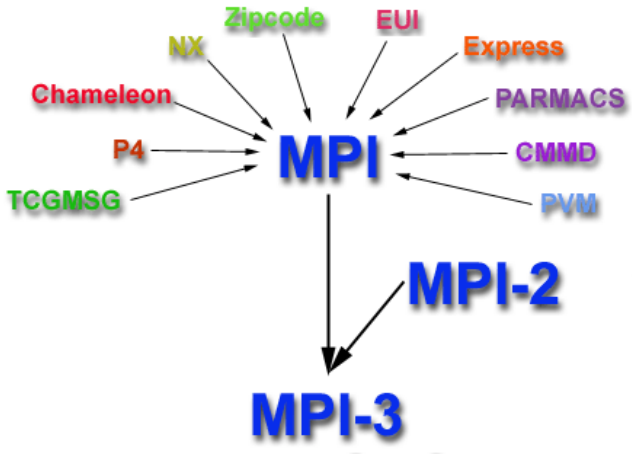

Figura 1.4: Evolução do MPI

A compatibilidade é preservada de uma versão inferior para uma versão superior. Por exemplo, um programa válido para a versão MPI-1.1 é válido também para as versões MPI-1.3 e MPI-2.2 do padrão; um programa válido na para a versão MPI-1.3 é válido também para a versão MPI-2.2 do padrão, e assim por diante.

A comunidade está sempre trabalhando na melhoria da especificação do MPI, revisando as definições herdadas das especificações anteriores e adicionando novas atualizações, com o intuito de manter a modernidade e aplicabilidade do padrão.

1.2.2 Objetivos

Resumidamente o objetivo central do MPI é oferecer uma interface padrão para o desenvolvimento de programas baseados no paradigma de troca de mensagens. Para isso, é necessário que as funções definidas sejam portáteis, flexíveis e permitam

implementações eficientes de acordo com as características do ambiente de execução escolhido.

Outros objetivos gerais do MPI são (MPI Forum 2009):

- Definir uma interface de programação de aplicações (API) — em vez de apenas uma interface para uso pelos compiladores ou uma implementação de biblioteca de sistema.
- Garantir uma comunicação eficiente:
    - evitando cópias de memória para memória;
    - permitindo superposição de comunicação e computação.
    - permitindo o uso de coprocessadores de comunicação, quando disponíveis.
- Permitir o uso do padrão proposto em ambientes heterogêneos.
- Facilitar o uso da interface por linguagens como C, C++ e Fortran.
- Assumir que a interface de comunicação é confiável:
    - falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.
- Definir uma interface que possa ser implementada em diversas plataformas, sem mudanças significativas no sistema de comunicação subjacente ou *software* de sistema.
- Tornar a semântica da interface independente de linguagem.
- Permitir o uso seguro de *threads* (fluxos de execução independentes dentro de um mesmo processo).

O padrão MPI é uma forma eficiente de implementação de programas paralelos. Embora esteja voltado para máquinas de memória distribuída, permite a sua utilização também em máquinas com memória compartilhada, com desempenho equivalente.

1.3 OpenMP

1.3.1 Histórico

O OpenMP Architecture Review Board (ARB) publicou suas primeiras especificações da API do OpenMP para Fortran em outubro de 1997. Em outubro do ano seguinte, eles lançaram o padrão para as linguagens C e C++. No ano 2000 foi lançada a versão 2.0 das especificações Fortran, seguindo-se em 2002 do lançamento da versão 2.0 das especificações para as linguagens C e C++. A versão 2.5 é uma especificação combinada para C, C++ e Fortran lançada em 2005.

Até a versão 2.0, o OpenMP especificava principalmente formas de paralelizar laços altamente regulares, do modo como ocorre em programação numérica orientada por matriz, onde o número de iterações do laço é conhecido no momento da entrada. Isso foi reconhecido como uma limitação e várias extensões paralelas de tarefas (*tasks*) foram adicionadas às implementações. Em 2005, foi formado um esforço para padronizar o paralelismo de tarefas, resultando na publicação de uma proposta em 2007, inspirada nos recursos de paralelismo de tarefas existentes nas linguagens Cilk, X10 e Chapel. (Ayguadé et al. 2008)

Esse trabalho resultou na versão 3.0, que foi lançada em maio de 2008. Os novos recursos desta versão incluem o conceito de tarefas e a construção de tarefas, (OpenMP Architecture Review Board 2008) ampliando significativamente o escopo do OpenMP além das

construções de laços paralelos, que eram a maior parte da especificação OpenMP 2.0.

A versão 4.0 da especificação foi lançada em julho de 2013. Além de melhorias relacionadas à construção e controle de tarefas, foram acrescidos ou melhorados os seguintes recursos (OpenMP Architecture Review Board 2013):

- suporte para aceleradores e *manycores*;
- suporte à movimentação de dados e à diretiva *atomic*;
- extensões de tarefas e manipulação de erros;
- opções de afinidade entre *threads*;
- operações de redução definidas pelo programador;
- suporte às instruções SIMD e ao Fortran 2003.

Pouco tempo depois, em novembro de 2015, foi lançada a versão 4.5 da especificação do OpenMP, cujas principais mudanças realizadas foram no suporte a aceleradores e *manycores*, principalmente em relação ao comportamento de um trecho de código quando executado em um dispositivo (OpenMP Architecture Review Board 2015).

- OpenMP versão 1.0 em 1997;
    - A primeira versão foi apenas para a linguagem FORTRAN, seguida das versões para C/C++ no ano seguinte.
- OpenMP versão 2.0 em 2000;
    - Até esta versão apenas laços altamente regulares eram paralelizados.
- OpenMP versão 3.0 em 2008;

    - Incluídos os conceito de tarefas e a construção de tarefas

- OpenMP versão 4.0 em 2013;
    - Nesta versão foram incluídos o suporte para aceleradores; reduções definidas pelo usuário e suporte ao SIMD.
- OpenMP versão 4.5 em 2015;
    - Nesta versão foram incluídos melhorias no suporte para aceleradores e redução em *arrays*.
- OpenMP versão 5.0 em 2018;
    - A última versão até a edição deste livro.

A versão atual, 5.0, foi lançada em novembro de 2018 (OpenMP ARB 2018). Nessa versão, diversas construções foram adicionadas, principalmente no controle de tarefas e seu comportamento, tanto no processador quanto nos aceleradores e *manycores*, além da redução em *arrays*. As áreas do OpenMP com mais melhorias foram o controle de dependências entre tarefas, o modelo de memória e seu respectivo gerenciamento. Deve-se notar que nem todos os compiladores (e sistemas operacionais) suportam o conjunto completo de recursos para a(s) última(s) versão(ões).

1.3.2 Objetivos

O OpenMP foi projetado para a programação de computadores paralelos com memória compartilhada. A facilidade principal é a existência de um único espaço de endereçamento através de todo o sistema de memória, assim cada processador pode ler e escrever em todas as posições de memória.

É possível a utilização do OpenMP em diversos tipos de arquitetura, quais sejam:

- Arquiteturas com memória compartilhada centralizada
- Arquiteturas com memória compartilhada distribuída
- Arquiteturas *manycore*
- Aceleradores

O paralelismo no OpenMP é obtido pela execução simultânea de diversas *threads* dentro das regiões paralelas. Haverá ganho real de desempenho se houver processadores disponíveis na arquitetura para efetivamente executar essas regiões em paralelo.

Por exemplo, as diversas iterações de um laço podem ser compartilhadas entre as diversas *threads* e, se não houver dependências de dados entre as iterações do laço, poderão também ser executadas em paralelo.

Os laços são a principal fonte de paralelismo em muitas aplicações. Se as iterações de um laço são independentes (podem ser executadas em qualquer ordem), então podemos compartilhar as iterações entre *threads* diferentes. Por exemplo, se tivermos duas *threads* e o seguinte laço:

```
for (i = 0; i<100; i++)
    a[i] = a[i] + b[i];
```

As iterações 0-49 podem ser feitas em uma *thread* e as iterações 50-99 na outra.

O OpenMP faz uso combinado de diretivas passadas para o compilador, assim como de funções definidas na sua biblioteca, para explorar o paralelismo no código em linguagem C, C++ ou Fortran. É possível também modificar o comportamento da execução de uma aplicação a partir de informações passadas pelas variáveis de ambiente de um sistema operacional.

1.4 OpenACC

1.4.1 Histórico

O OpenACC foi projetado de modo que os cientistas e engenheiros possam portar seus códigos de aplicação científica (HPC) para uma ampla variedade de plataformas e arquiteturas de *hardware* heterogêneas. É um modelo de programação paralela portátil, baseado em diretivas definidas pelo usuário, e requer um esforço muito menor de programação do que os modelos de baixo nível. A especificação OpenACC oferece suporte às linguagens de programação C, C++, Fortran e várias arquiteturas de *hardware*, incluindo processadores x86 e POWER e, principalmente, aceleradores baseados em GPUs da NVIDIA e AMD.

Uma das primeiras versões do padrão, o OpenACC 2.0, foi lançada oficialmente em junho de 2013 (OpenACC-Standard.org 2013). Seguiram-se então diversas atualizações do padrão: a versão 2.5 da especificação foi lançada em outubro de 2015, enquanto a versão 2.6 foi lançada em novembro de 2017, seguida da versão 2.7, em novembro de 2018. Já a versão 3.1 da especificação, apresentada neste livro, está disponível desde novembro de 2020 (OpenACC-Standard.org 2020).

1.4.2 Objetivos

O OpenACC é um modelo de programação aberta para computação paralela desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore*, *manycore* e GPUs.

O OpenACC é compatível com os modelos de programação OpenMP e MPI, ambas as abordagens podem ser combinadas com o OpenACC. Em geral, as diretivas do OpenACC são muito

semelhantes às do OpenMP. Em relação ao CUDA, OpenACC é totalmente compatível tornando a necessidade de alteração do código a menor possível.

As seguintes características podem ser destacadas no modelo de programação OpenACC:

- Fácil e simples de utilizar;
- Independente de fabricante;
- Oculta a complexidade de *hardware* dos programadores;
- Requer poucas modificações no código fonte;
- Mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- O mesmo código, sem modificações, por ser usado em arquiteturas *multicore, manycore* e GPUs;
- É similar ao OpenMP (familiaridade);
- Permite fácil transição para o OpenMP 4.5.

---

1. Fonte: https://computing.llnl.gov/tutorials/mpi/images/MPIevolution.gif ↩

2 Conceitos básicos

Neste capítulo são apresentados alguns conceitos básicos sobre a estrutura de processos e *threads* no sistema operacional, modelos básicos de arquiteturas paralelas e os paradigmas de programação paralela associados. São discutidos também os fundamentos sobre a criação; o particionamento de dados e tarefas; a sincronização; e o balanceamento de carga em programas paralelos. Finalmente, métricas e métodos de avaliação do desempenho de programas paralelos são descritos.

2.1 Processos e Threads

Nos sistemas de computação modernos, as aplicações do usuário usam os serviços fornecidos pelo sistema operacional para acessar os recursos físicos da máquina (memória principal, memória secundária, processador e dispositivos de E/S). O sistema operacional é uma camada de *software* interposta entre o *hardware* e as aplicações, como ilustrado na Figura 2.1, e tem dois propósitos principais: proteger o *hardware* de erros (ou mau uso) das aplicações (Tanenbaum and Woodhull 2006); e prover mecanismos para simplificar o acesso aos recursos físicos da máquina (Stallings 2009).

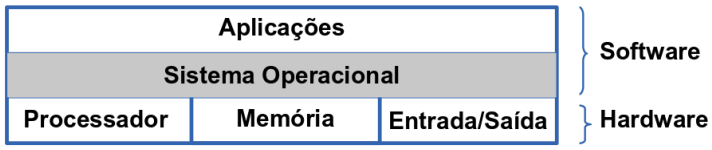

Figura 2.1: Visão em camadas de um sistema de computação.

Para isso, os sistemas operacionais oferecem três abstrações fundamentais: arquivos, memória virtual e processos. A Figura 2.2 mostra a relação entre essas abstrações.

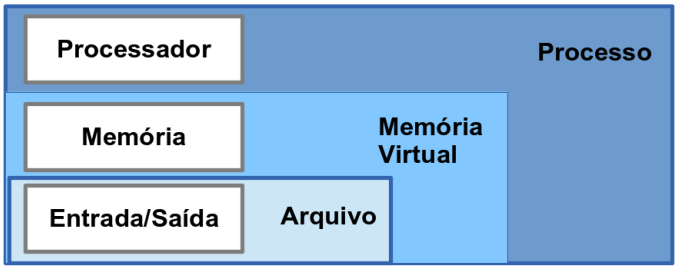

Figura 2.2: Abstrações providas pelo sistema operacional

A abstração de arquivo simplifica o acesso aos dispositivos de E/S e permite armazenar dados de forma permanente. A abstração de memória virtual gerencia o acesso à memória principal e secundária. Por fim, a abstração de processo encapsula todos os recursos necessários à execução de um programa na máquina (Bryant and David Richard 2010), podendo conter também vários fluxos independentes de execução, chamados de *threads*.

2.1.1 Processos

Quando um programa é executado no computador, o sistema operacional é o responsável por prover o acesso a todos os recursos de *hardware* necessários (tempo de processador, espaço de memória e alocação dos dispositivos de E/S). Como vários programas podem estar ativos ao mesmo tempo, o sistema operacional usa a abstração de processo, ilustrada na Figura 2.3, para gerenciar o acesso concorrente aos recursos da máquina e prover a ilusão de uso exclusivo desses recursos por cada programa (Tanenbaum and Woodhull 2006).

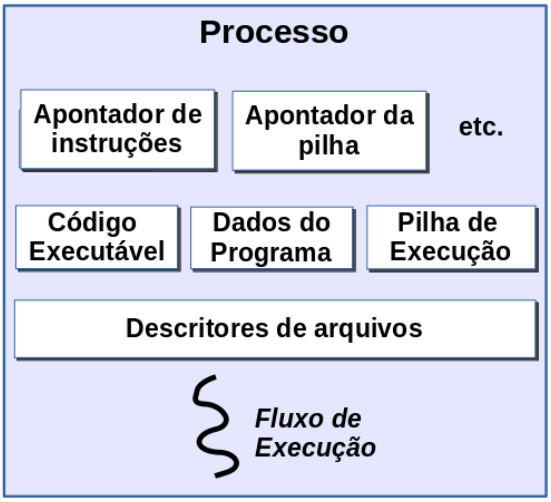

Figura 2.3: Informações de um processo

Assim, todo processo possui um espaço de endereçamento privado na memória onde são armazenados o código e os dados do programa e sua pilha de execução (espaço de memória usado para as chamadas de função e passagem de parâmetros). Além disso, cada processo está associado a um conjunto de registradores. Entre eles, temos o apontador de instruções (mantém o endereço da próxima instrução a ser executada), o apontador de pilha (armazena o endereço do topo da pilha) e várias outras informações e recursos necessários para a execução do programa.

Quando o sistema operacional executa um programa, um novo processo é criado e o arquivo executável do programa é carregado no contexto desse processo. Esse processo é executado como se o processo tivesse uso exclusivo do processador, quando na realidade ele é compartilhado entre diversos fluxos de execução de processos diferentes.

A Figura 2.4 ilustra esse conceito, onde a utilização de um mesmo processador é compartilhada por três fluxos de controle distintos, um para cada processo em execução, que se revezam no uso do processador. Cada processo executa uma porção do seu fluxo durante um intervalo de tempo predeterminado ou até solicitar uma

operação de E/S para o sistema operacional. Em um caso ou no outro, o processo atual é suspenso temporariamente, enquanto um novo processo passa a ser executado pelo processador.

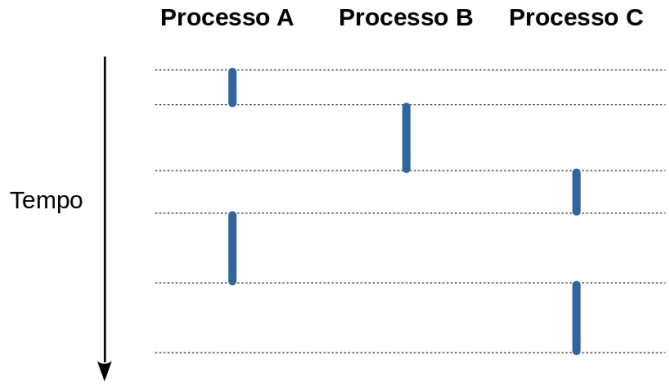

Figura 2.4: Alternância na execução dos processos ao longo do tempo

O sistema operacional é responsável por gerenciar a alternância entre processos, dando uma fatia determinada de tempo do processador para cada processo, garantindo que todos os processos evoluam e completem sua execução. Para que tudo isso seja possível, cada processo precisa manter informações sobre o estado da sua execução ao longo do tempo, até completar sua execução. Essas informações são mantidas dentro do **contexto do processo** e incluem:

- Código do programa;
- Dados armazenados na memória;
- Pilha de execução;
- Conteúdo dos registradores;
- Variáveis de ambiente; e
- Descritores de arquivos abertos.

Quando o sistema operacional decide transferir o uso do processador para outro processo, é executada uma sequência de ações, denominada **troca de contexto**, envolvendo:

- Salvar o contexto do processo atual;
- Restaurar o contexto do novo processo; e
- Passar o controle para o novo processo.

O novo processo inicia ou retoma sua execução exatamente do ponto onde parou anteriormente. A Figura 2.5 ilustra esse procedimento. A troca de contexto entre processos é normalmente um procedimento lento, exigindo uma chamada ao sistema operacional, com salvamento de todas as informações do contexto do processo atual, além da restauração das informações do contexto do novo processo a ser executado.

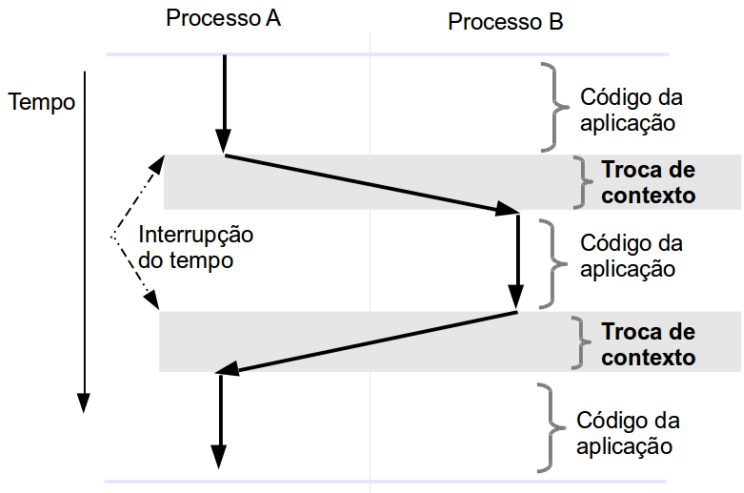

Figura 2.5: Troca de contexto entre processos

Nos sistemas operacionais modernos, um processo pode criar outros processos, chamados de processos "filhos". Os processos filhos possuem um espaço de endereçamento próprio e recebem inicialmente uma cópia do código e da pilha do processo pai. Como a maioria dos sistemas de computação modernos são equipados com múltiplos processadores, a criação de processos filhos permite um melhor aproveitamento dos recursos, com execução simultânea nos diversos processadores.

Esse é um dos princípios básicos da programação paralela: dividir a tarefa central de uma aplicação entre fluxos lógicos de execução independentes, visando reduzir o tempo total de execução da

aplicação (Rauber and Rünger 2013). Nesse caso, cada fluxo lógico dentro da aplicação é criado e mantido como um novo processo.

A programação paralela com o uso de vários processos é, portanto, uma das alternativas básicas para explorar os recursos paralelos da máquina com o objetivo de melhorar o desempenho das aplicações. Entretanto, como destacado anteriormente, a troca de contexto entre processos tem um custo elevado. Além disso, como o espaço de endereçamento de cada processo é separado, quando houver a necessidade de interação entre os diferentes fluxos de execução, será preciso usar algum mecanismo de comunicação interprocessos oferecido pelo sistema operacional.

O principal uso de processos para executar as diversas tarefas associadas a um programa paralelo ocorre em um cenário de sistemas distribuídos, onde cada processador possui sua própria memória e a comunicação entre processos é feita com troca de mensagens, como veremos em mais detalhes na Seção 2.2.2, evitando o custo associado à troca de contextos.

Contudo, quando os processos de uma aplicação paralela compartilham uma mesma memória global, encontramos uma melhor solução com o uso das *threads*, como veremos a seguir.

2.1.2 *Threads*

Muitos dos sistemas operacionais modernos suportam o conceito de *threads*, ou seja, um fluxo lógico de execução independente dentro de um mesmo processo (Silberschatz, Galvin, and Gagne 2013).

As *threads* de um processo compartilham o mesmo espaço de endereçamento de memória, arquivos abertos e outros recursos que compõem o contexto global do processo, como ilustrado na Figura 2.6. Cada *thread*, no entanto, tem seu próprio contexto de execução, o qual inclui o identificador único da *thread*, o estado da pilha de execução e os valores no apontador de instrução e nos

registradores de uso geral. Aplicações com mais de uma *thread* são normalmente chamadas aplicações *multithreading*, e são uma alternativa à programação paralela utilizando vários processos.

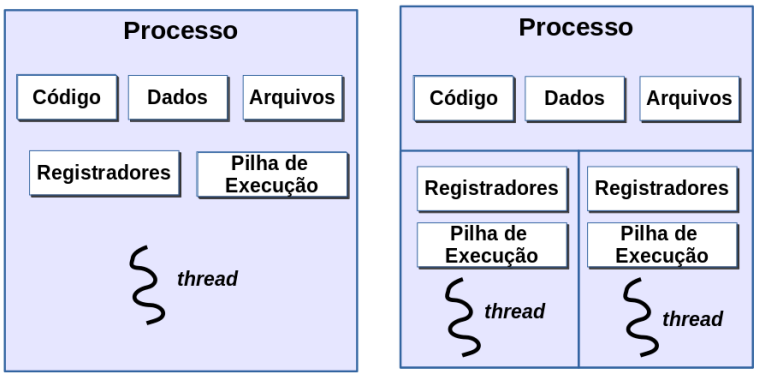

Figura 2.6: Processos com uma única *thread* e com várias *threads*

No modelo de execução com várias *threads*, todo processo começa a executar a partir de uma *thread* principal e, em algum ponto do código, a *thread* principal cria uma ou mais *threads*. Quando uma nova *thread* é criada, ela pode executar concorrentemente com as demais *threads* do processo. Entretanto, a execução das *threads* difere da execução de processos em alguns pontos importantes:

- As *threads* **não** são organizadas de forma hierárquica, como no caso de processos (processos pai e filhos). Todas as *threads* associadas a um processo formam um grupo de *threads* de mesmo nível, independente da ordem de criação ou de quais *threads* foram criadas por quais outras *threads*.

- A *thread* principal distingue-se das demais apenas por ser sempre a primeira *thread* a ser executada e, dependendo da implementação da biblioteca de gerência de *threads* usada, o

processo encerra automaticamente quando a *thread* principal termina.

O contexto específico de uma *thread* é semelhante ao contexto de uma função e, consequentemente, a troca de contexto entre *threads* implica no salvamento e recuperação de contextos relativamente leves, a exemplo do que ocorre em uma chamada de função dentro de um programa. Esse fato é o principal atrativo em favor do uso da programação com várias *threads* em vez de vários *processos*. Além disso, a comunicação entre as tarefas da aplicação é grandemente simplificada em uma implementação baseada em *threads*. Nesse caso, as tarefas compartilham o espaço de endereçamento de memória do processo que engloba as *threads*, eliminando a necessidade do uso de mecanismos especiais, mais restritos e, usualmente, mais lentos de comunicação interprocessos oferecidos pelos sistemas operacionais.

2.1.3 *Threads* de usuário e de núcleo

As *threads* podem ser implementadas dentro do espaço de usuário ou do espaço do núcleo do sistema do operacional. Em alguns sistemas, *threads* de usuário são chamadas simplesmente de *threads* e *threads* de núcleo são referidas como processos leves (LWP - do inglês *lightweight process*). Quando as *threads* são implementadas no núcleo, as trocas de contexto das *threads* são executadas pelo sistema operacional sem o conhecimento do aplicativo. Da mesma forma, quando as *threads* são implementadas no espaço de usuário, as trocas de contexto das *threads* são executadas por uma biblioteca de aplicativos, sem o conhecimento do sistema operacional. Há também abordagens híbridas nas quais, normalmente, uma biblioteca de espaço de usuário vincula uma ou mais *threads* de usuário a uma ou mais *threads* de núcleo (Tanenbaum and Woodhull 2006).

As *threads* de núcleo são suportadas e gerenciadas diretamente pelo sistema operacional. Uma *thread* de núcleo é uma entidade escalonável, ou seja, o escalonador do sistema operacional conhece e gerencia todas as *threads* de núcleo. Existe um bloco de controle de processo (PCB - do inglês *Process Control Block*) para cada processo ativo no sistema operacional, assim como um bloco de controle de *thread* (TCB - do inglês *Thread Control Block*) para cada *thread* no sistema operacional. Uma *thread* de núcleo executa dentro de um processo, mas pode ser referenciada por uma outra *thread* no sistema operacional.

Vantagens da *thread* de núcleo:

- O sistema operacional tem completo conhecimento de todas as *threads*;
- O escalonador pode decidir dar mais tempo de processador a um processo com um alto número de *threads*;
- É bom para aplicações que realizam tarefas, como operações de E/S, capazes de bloquear a *thread* com frequência.

Desvantagens da *thread* de núcleo:

- O sistema operacional gerencia e escalona todas as *threads*;
- Há uma sobrecarga significativa e um aumento na complexidade das rotinas do sistema operacional;
- Uma mudança do modo de execução do processador, de normal para privilegiado, é necessária para transferir o controle de uma *thread* para outra, em um mesmo processo;
- As *threads* de núcleo são mais lentas para serem criadas e gerenciadas quando comparadas com as *threads* de usuário.

Uma *thread* de usuário é uma entidade usada por programadores para lidar com múltiplos fluxos de controle dentro de um programa. Uma *thread* de usuário existe apenas dentro de um processo, e as *threads* em um processo não podem referenciar *threads* em outro processo.

Vantagens da *thread* de usuário:

- Podem ser implementadas em um sistema operacional que não suporte *threads* de núcleo;
- Não necessitam de modificações no sistema operacional;
- Possuem uma representação simples: apontador de instruções, registradores, pilha e um pequeno TCB são todos armazenados no espaço de endereçamento do processo no nível de usuário;
- São de gerenciamento simples: a criação, troca e sincronização de *threads* pode ser feita em nível de usuário sem intervenção do sistema operacional;
- São rápidas e eficientes: o custo da troca de contexto entre *threads* é similar a uma chamada de função.

Desvantagens da *thread* de usuário:

- Há falta de coordenação entre o gerenciador de *threads* em nível de usuário e o escalonador do sistema operacional;
- Um processo pode ser completamente bloqueado quando uma de suas *threads* fica bloqueada, mesmo havendo outras *threads* que o processo possa executar;
- Dar a um processo a mesma fatia de tempo de uso do processador, independente de o processo ter 1 ou 1000 *threads*;

- Deixar de executar outras *threads* de um processo quando uma de suas *threads* está de posse de uma trava (*lock*);

- Pode requerer a comunicação entre o gerenciador de *threads* do sistema operacional e o nível de usuário para superar os problemas aqui relatados.

As *threads* de usuário geralmente são mais portáteis e podem executar trocas de contexto mais rápido e com menor custo, quando comparadas com as *threads* de núcleo. Por outro lado, as *threads* de núcleo podem aproveitar as máquinas com múltiplos processadores e não possuem nenhum problema de bloqueio de outras *threads* do mesmo processo quando são realizadas operações de E/S. As *threads* de núcleo geralmente são escalonadas de modo preemptivo lado a lado com os processos subjacentes, sendo que as *threads* de usuário podem usar tanto o escalonamento preemptivo como o não preemptivo.

Escalonamento preemptivo de *threads* vs. não preemptivo

No escalonamento preemptivo, o escalonador permite a execução de uma *thread* até que ocorra uma situação de bloqueio (geralmente uma chamada de função bloqueante) ou após decorrido o intervalo de tempo (*timeslice*) atribuído para a execução da *thread* pelo processador. Em seguida, o escalonador retira o controle da *thread* sem nenhuma chance de oposição da *thread*. Isso geralmente é realizado interrompendo a *thread* por meio de um sinal de interrupção de *hardware* (para *threads* de núcleo) ou um sinal de interrupção de *software* (para *threads* de usuário).

No escalonamento não preemptivo, após uma *thread* receber o controle do processador pelo escalonador, a execução da *thread* continua até que ocorra uma situação de bloqueio (novamente, uma chamada de função bloqueante que, em vez disso, retorna de volta ao escalonador), ou até que a *thread* devolva explicitamente o controle ao escalonador, de uma forma cooperativa.

2.2 Programação paralela

De forma geral, uma aplicação é desenvolvida com um único fluxo de execução. Naturalmente, a execução acontecerá de forma tradicional em apenas um processador. Dizemos então que foi utilizada uma programação sequencial para a construção dessa aplicação. Para que a aplicação faça um melhor uso dos recursos de *hardware* existentes em uma máquina é importante conhecer o funcionamento das arquiteturas paralelas e consequentemente fazer uso da programação paralela de uma forma mais eficiente (Rauber and Rünger 2013). Maiores detalhes sobre as arquiteturas paralelas podem ser vistos na Seção 2.5.

2.2.1 Programação com Memória compartilhada

Nesse paradigma de programação, a comunicação entre as tarefas é realizada por meio de variáveis armazenadas em um espaço de memória compartilhado. As tarefas da aplicação podem ser mapeadas em processos ou *threads*.

No caso de processos, é necessário requisitar ao sistema operacional a definição de um espaço de memória compartilhado por mais de um processo. Este compartilhamento entre processos pode ser complexo e pouco eficiente.

No caso do uso de *threads*, já existe um espaço de memória compartilhado entre as *threads* no mesmo processo que as criou. Esse acesso é rápido e de baixa complexidade. Por isso, o uso de *threads* é frequente no paradigma de memória compartilhada.

Quando duas ou mais *threads* de um programa precisam trocar informações entre si, o programador define variáveis de acesso compartilhado entre elas. Desse modo, quando uma *thread* tem um

valor novo para ser enviado para as outras *threads*, esse valor é simplesmente escrito em uma variável compartilhada, independentemente do fato de as demais *threads* estarem esperando por esse valor naquele momento.

Quando a outra *thread* precisa saber qual é o valor atual dessa informação, o conteúdo atual da variável compartilhada é simplesmente lido, como ilustrado na Figura 2.7. Dessa forma, toda a comunicação entre as *threads* se dá de forma assíncrona, ou seja, as *threads* escrevem/lêem valores nas variáveis compartilhadas a qualquer tempo.

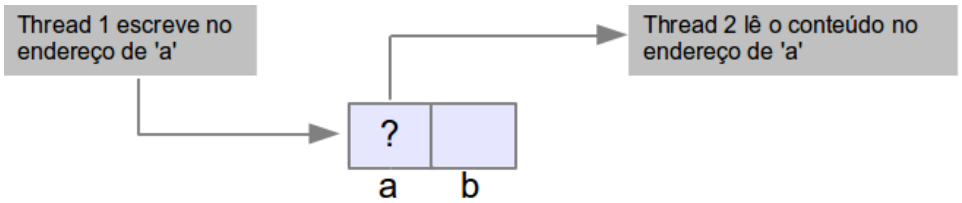

Figura 2.7: Comunicação assíncrona entre *threads* via memória compartilhada

Isso pode, evidentemente, levar a uma diferença entre os valores observados pelas diversas *threads* quando executando simultaneamente, em contraposição a uma execução sequencial. Antes de nos aprofundarmos nos detalhes de como resolver esse problema, vamos repassar alguns conceitos importantes relativos à programação paralela.

Concorrência e paralelismo

Infelizmente, os adjetivos "concorrente" e "paralelo" são usualmente empregados como sinônimos um do outro e também usados de maneiras distintas por autores diferentes. Mas há, efetivamente, uma diferença entre eles. Segundo (Scott 2013): "duas operações são *concorrentes* se ambas foram iniciadas e nenhuma foi ainda completada; duas operações são *paralelas* se realmente podem ser

executadas ao mesmo tempo. O paralelismo é, portanto, uma forma de implementação da concorrência."

No escopo deste livro, duas operações são consideradas *concorrentes* se foram iniciadas, e não foram ainda completadas, como no caso da sua execução alternada no tempo por apenas um servidor. Elas serão consideradas *paralelas* se realmente forem executadas simultaneamente no tempo, como no caso de haver mais de um servidor que permita executá-las de forma simultânea. Logo, duas operações paralelas são necessariamente concorrentes, mas duas operações concorrentes não são necessariamente paralelas.

Quando falamos de paralelismo ou concorrência, algumas definições são importantes para serem lembradas:

- Condição de Corrida – Situação onde ocorre a manipulação simultânea de um recurso por duas ou mais *threads*.

- Seção ou Região Crítica – Trecho de código que coordena o acesso a um recurso compartilhado.

- Exclusão mútua – Propriedade do *software* que assegura acesso exclusivo a um recurso compartilhado.

- Deadlock – Condição especial criada por dois ou mais processos (*threads*) e duas ou mais travas de recurso que impedem os processos (*threads*) de realizar trabalho produtivo.

Ao longo das demais seções vamos procurar detalhar melhor esses termos e suas respectivas definições.

Sincronização por exclusão mútua

A comunicação assíncrona entre as *threads* pode gerar **condições de corrida** no programa. Trata-se de um problema típico da programação concorrente, percebido quando duas ou mais *threads*

tentam ler e escrever simultaneamente em uma mesma variável e o resultado da computação depende da ordem de execução das *threads*.

Para que uma comunicação entre *threads*, via memória compartilhada, possa ocorrer de forma correta, isto é, sem gerar condições de corrida indesejáveis, é necessário usar mecanismos de sincronização por exclusão mútua entre as *threads*.

A **sincronização por exclusão mútua** visa garantir que os trechos de código em cada *thread*, acessando variáveis compartilhadas, não sejam executados ao mesmo tempo. Apenas uma das *threads* pode executar o trecho de código protegido pela operação de exclusão mútua. Essa restrição é necessária para evitar a inconsistência nos valores das variáveis compartilhadas.

Como exemplo, podemos tomar o caso de uma operação que incrementa o valor de uma variável global "a" (*ex., a++;*). Para realizar esse incremento, três instruções de nível mais baixo (linguagem de máquina) são normalmente necessárias:

1. ler o valor atual da variável;
2. incrementar esse valor;
3. escrever o novo valor na variável global.

Se duas *threads* executam essa operação concorrentemente, as duas poderão ler o mesmo valor inicial de *a* e realizar o incremento sobre esse valor. Ao final, embora duas operações de incremento tenham sido realizadas, o valor em *a* refletirá apenas um incremento. Com o uso da operação de exclusão mútua apenas uma das *threads* tem acesso exclusivo à variável compartilhada, realizando o seu incremento sem interferência da outra *thread*.

O uso correto da programação paralela requer a proteção de regiões críticas, de modo a evitar condições de corrida no código. Normalmente, não é seguro o acesso simultâneo por várias *threads* ao mesmo recurso. Esse trecho de código que faz acesso ao recurso

compartilhado é chamado de seção ou região crítica. A proteção da região crítica é fundamental para evitar a ocorrência de concorrência insegura e de condições de corrida.

As condições de corrida podem ocorrer em um sistema concorrente, quando as *threads* e processos são escalonados entremeados no tempo, ou em um computador paralelo, na qual dois processos (ou *threads*) estão tentando entrar em uma região crítica ao mesmo tempo.

É necessário assegurar que apenas uma *thread* manipule os recursos compartilhados de cada vez. Portanto, é necessário um mecanismo para impedir o acesso a um recurso enquanto outra *thread* o utiliza. Os mecanismos reais usados para fornecer sincronização e proteção contra condições de corrida são:

1. **Operações atômicas:** operações atômicas fornecem instruções que são executadas atomicamente, isto é, sem interrupção. Operadores atômicos são instruções indivisíveis capazes de executar a leitura e escrita de memória em uma única etapa indivisível e ininterrupta.

2. **Spin Locks:** a ideia por trás do *spin lock* é executar um laço ou ficar em espera ocupada até que a trava (*lock*) seja liberada. Geralmente são utilizadas instruções atômicas para garantir acesso indivisível à posição de memória correspondente à variável da trava. Um *spin lock* pode ser mantido por no máximo uma *thread* e é usado para travas mantidas por períodos curtos.

3. **Semáforos:** semáforos ou semáforos de contagem são travas que colocam o processo para dormir, limitando o número máximo de processos que possuem a trava simultaneamente. Quando um processo tenta adquirir um semáforo que não está disponível, o semáforo coloca o processo em uma fila de espera e suspende o processo, colocando-o para dormir. O processador fica então livre para executar outra *thread* ou processo. Mais

tarde, quando o semáforo se torna disponível, um dos processos na fila de espera é despertado para adquirir o semáforo. Por causa de sua natureza dormente, os semáforos são adequados para travas mantidas por um longo tempo.

Deadlock

Mas alguns problemas podem surgir do uso desses mecanismos de sincronização. Se as condições de corridas são causadas por falta de sincronização, os *deadlocks* são causados por sincronização excessiva ou incorreta. Um *deadlock* ocorre quando cada tarefa em um conjunto de tarefas é bloqueada à espera de um recurso pertencente a outra tarefa no conjunto. Todas as condições apresentadas na Tabela 2.1 devem ser verdadeiras simultaneamente para a ocorrência de um *deadlock*.

Tabela 2.1: Condições necessárias para ocorrência de deadlock

Exclusão mútua	Existem recursos que não podem ser usados simultaneamente por mais de uma tarefa.
Obter e esperar	Tarefas que já possuem um recurso podem bloquear à espera por outro recurso.
Nenhuma preempção	Recursos não podem ser removidos à força da tarefa que os possui. Somente a tarefa proprietária pode liberar recursos por si só.
Espera circular	Uma tarefa está aguardando um recurso que está sendo mantido por outra tarefa, que, por sua vez, aguarda a primeira tarefa liberar um recurso que ela mantém.

A remoção de qualquer uma das condições listadas é suficiente para evitar *deadlocks*. Em outras palavras:

- A condição de exclusão mútua pode ser removida, limitando-se a codificação ao uso de um única trava.

- As condições de obter e esperar podem ser evitadas se não mais de uma trava/recurso for adquirido por vez e também não se realizar a aquisição de travas recursivamente.

- A condição de espera circular pode ser evitada se as travas/recursos forem sempre adquiridos na mesma ordem e liberados na ordem inversa para evitar ciclos.

2.2.2 Programação por Troca de mensagens

Nesse paradigma de programação, a comunicação entre os diferentes fluxos de execução da aplicação é feita por meio de primitivas básicas para envio e recebimento de mensagens. Como não há um espaço de memória compartilhada, as tarefas são normalmente mapeadas em processos, cada um com sua memória privada, e o problema de condição de corrida não ocorre, embora também possa haver *deadlock*. A Figura 2.8 ilustra essa forma de comunicação. Maiores detalhes sobre como evitar a ocorrência de *deadlock* no contexto da programação por troca de mensagens podem ser vistos na Seção 3.3.6.

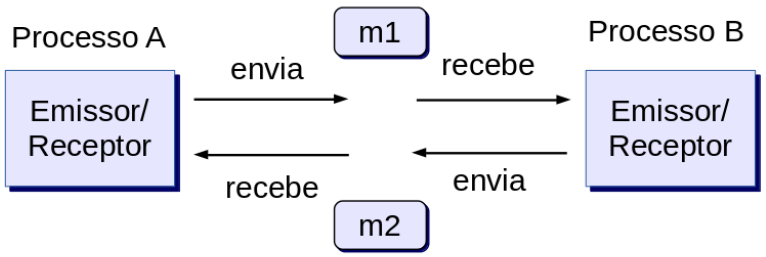

Figura 2.8: Comunicação entre processos via troca de mensagens

Há dois modos principais de comunicação no paradigma de programação por troca de mensagens: síncrono e assíncrono. Nas subseções seguintes são apresentados detalhes sobre cada um deles.

Modo de comunicação síncrona

Nesse modo de comunicação, o processo de origem da mensagem faz uma requisição inicial de autorização de envio ao processo destinatário. Depois de receber essa autorização, a operação de envio (*send*) é realizada. A operação de envio só se completa quando a operação de recepção (*receive*) correspondente retorna uma mensagem de confirmação (*acknowledgement*). Esse modelo de comunicação requer, portanto, um protocolo de quatro fases para a sua implementação, como ilustrado na Figura 2.9 [1]:

1. pedido de autorização para transmitir;
2. recebimento da autorização;
3. transmissão da mensagem;
4. recebimento da mensagem de reconhecimento.

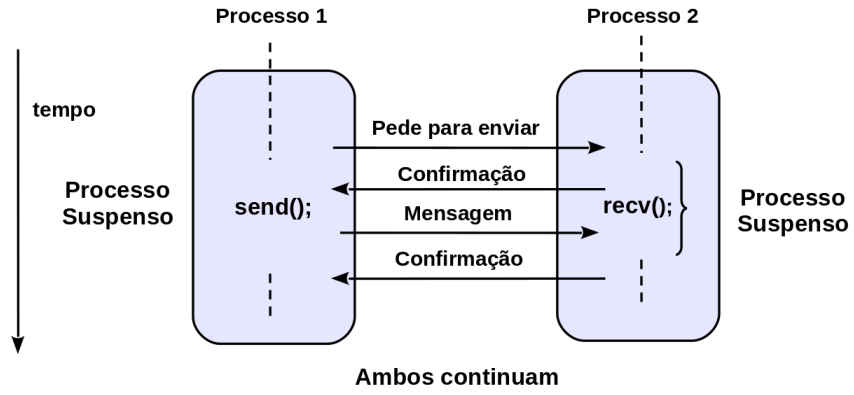

Figura 2.9: Comunicação completamente síncrona

No modo de comunicação síncrona, tanto o envio como a recepção de mensagens são bloqueantes. A função de envio não retorna para o programa principal enquanto a mensagem não for completamente enviada. E a função de recepção também não retorna enquanto a mensagem não estiver disponível para o usuário no espaço de

memória reservado na aplicação para isso. Como consequência, ao se concluir a operação de envio, as estruturas de dados utilizadas no programa do usuário para o envio da mensagem podem ser reutilizadas com segurança. Ao término da operação de recepção temos a certeza de que os dados recebidos estão disponíveis para a aplicação.

Esse modo de comunicação é simples e seguro, contudo elimina a possibilidade de haver superposição entre o processamento da aplicação e a transmissão das mensagens, diminuindo as possibilidades de exploração de paralelismo.

Modo de comunicação assíncrona

A principal característica da comunicação assíncrona é a desvinculação do instante de tempo no qual ocorrem as operações de envio e recepção da mensagem. Isso se dá principalmente com o uso de áreas (*buffers*) destinadas ao armazenamento da mensagem enviada, enquanto a transmissão dos dados é realizada através da rede de comunicação.

Assim sendo, a operação de envio (*send*) se completa a partir da cópia de todo o conteúdo da mensagem do espaço de endereçamento do usuário para uma área correspondente (*buffer*) da biblioteca ou do sistema operacional. Isso libera as estruturas de dados do programa do usuário para serem utilizadas por novas operações de envio de mensagens, embora não haja nenhuma garantia que a transmissão da mensagem tenha completado ou vá completar satisfatoriamente. Esse mecanismo é ilustrado na Figura 2.10 [2].

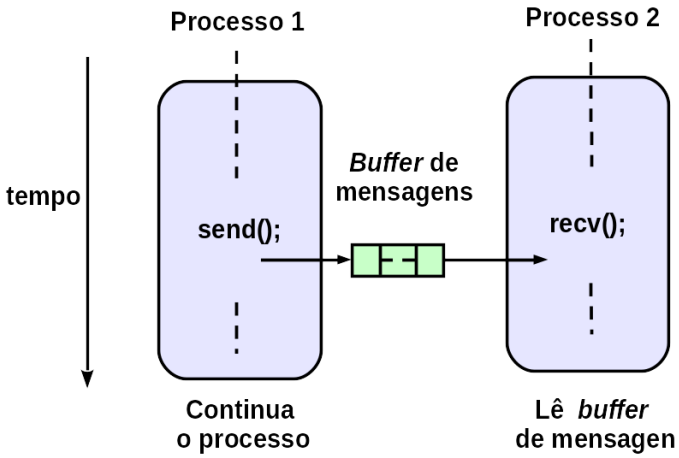

Figura 2.10: Comunicação assíncrona

Mas não é apenas isso, há outras formas de implementação da comunicação assíncrona, como o uso de rotinas de envio e/ou recepção não bloqueantes.

Quando a operação de envio é não bloqueante, ela retorna imediatamente, dando apenas início ao processo de transmissão da mensagem. Isso nada garante e não autoriza a reutilização das estruturas de dados do programa para o envio de uma nova mensagem. Quando a operação de recepção é não bloqueante, ela apenas indica a intenção de realizar uma operação de recepção, retornando também imediatamente, e nada se garante em relação aos dados recebidos já estarem disponíveis para uso pela aplicação do usuário.

Tanto nas operações de envio como de recepção não bloqueantes, ainda é necessário uma segunda fase, onde é verificado se as operações de troca de mensagem iniciadas anteriormente pelas primitivas de envio e recepção já foram concluídas. Somente após essa verificação é possível alterar com segurança a área de dados utilizada pelo processo emissor e ter certeza de que os dados recebidos estão disponíveis para uso no processo receptor da mensagem.

O modo de comunicação assíncrono permite uma maior superposição no tempo entre o processamento da aplicação e a transmissão das mensagens, aumentando as possibilidades de exploração de paralelismo.

2.2.3 Sincronização de barreira

A sincronização refere-se aos mecanismos que permitem ao programador controlar a ordem relativa na qual as operações ocorrem em diferentes *threads* ou processos da aplicação. Na Seção 2.2.1, a sincronização por exclusão mútua foi descrita, em conjunto com a necessidade do seu uso na programação com memória compartilhada. Nesta seção, outro mecanismo de controle é apresentado: a **sincronização de barreira**, necessária tanto na programação com memória compartilhada como na programação por troca de mensagens.

A sincronização de **barreira** é utilizada quando o programa é organizado em várias etapas e as tarefas, quer estejam mapeadas para *threads* ou processos, não podem avançar para a etapa seguinte enquanto todas as tarefas não finalizarem a etapa atual, como ilustrado na Figura 2.11. Nesta figura, as tarefas chegam ao ponto de sincronização em momentos distintos. Todas as tarefas ficam bloqueadas (**A, B e C**) esperando a última tarefa (**D**) alcançar esse ponto, quando então todas as tarefas são liberadas para prosseguirem a sua execução.

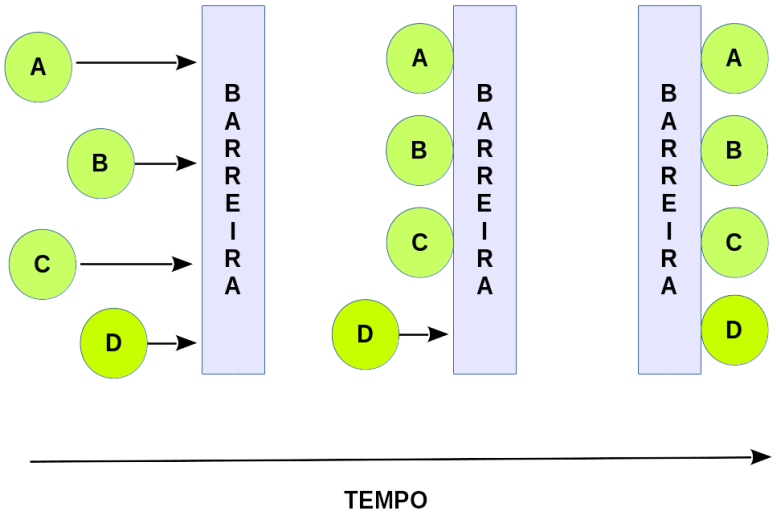

Figura 2.11: Sincronização com barreira

No modelo de programação com troca de mensagens, os mecanismos de sincronização podem ser embutidos nas próprias primitivas básicas de comunicação. Quando uma mensagem é enviada de um processo para outro, é possível determinar o bloqueio (espera) do processo emissor até que o processo receptor esteja apto para receber a mensagem; ou que o processo receptor fique bloqueado até que o processo emissor gere e envie a informação esperada. Além disso, assim como no paradigma de programação em memória compartilhada, também é possível oferecer operações coletivas de sincronização, tais como barreiras.

Existem diferentes mecanismos para implementar essa forma de sincronização entre *threads* ou processos, dependendo do paradigma de programação e da biblioteca de programação paralela utilizados, que serão abordados nos capítulos seguintes deste livro.

2.2.4 Particionamento de dados e tarefas

Um dos primeiros passos para a criação de um programa paralelo é quebrar o problema em partes de trabalho para serem distribuídas aos diferentes fluxos de execução (tarefas). Essa atividade é conhecida como decomposição ou particionamento de dados e

tarefas. Nesse passo, buscamos extrair o máximo de paralelismo sem gerar uma sobrecarga significativa de gerenciamento das tarefas em relação ao trabalho útil a ser feito. Ou seja, as tarefas de comunicação e sincronização não podem ser mais complexas do que as tarefas de computação. Se isso ocorrer, a execução em paralelo certamente será mais ineficiente do que a sequencial. Tipicamente, o particionamento:

- depende do modelo de programação adotado e da eficiência das primitivas de comunicação e sincronização desse modelo na plataforma alvo;
- explora a localidade dos dados;
- objetiva a redução da serialização no acesso a recursos compartilhados;
- procura a redução do custo de comunicação e da sincronização visto pelos processadores;
- visa escalonar as tarefas com maior cadeia de dependência para serem executadas mais cedo.

Depois de modelados, os diferentes fluxos de execução (tarefas) deverão ser alocadas a processos ou *threads* de modo que:

- haja um balanceamento de carga adequado e a redução do tempo de espera por conta da sincronização;
- possa haver uma redução da comunicação entre processos ou *threads*;
- a sobrecarga do gerenciamento seja a mais reduzida possível.

Os processos e *threads* serão escalonados para execução nos processadores reais disponíveis no sistema. Para realizar isso de uma maneira eficiente, devemos explorar a localidade dos dados,

mantendo os processos relacionados alocados ao mesmo processador. No caso de programas com memória distribuída, se a rede de interconexão entre as máquinas apresentar custos de comunicação variáveis, sugere-se alocar os processos com maior carga de comunicação aos processadores com menor custo de comunicação entre si.

Isso posto, há duas maneiras básicas de particionar o trabalho computacional entre tarefas: decomposição do domínio de dados ou decomposição funcional.

- Decomposição do domínio de dados

  Neste tipo de particionamento, os dados associados com o problema são decompostos. Cada tarefa trabalha então sobre uma parcela dos dados. A decomposição pode ser cíclica, com pequenas parcelas de dados para cada tarefa, ou em blocos, com o domínio total dividido em grandes blocos para cada tarefa.

  Exemplo: multiplicação de matrizes (densas ou esparsas)

- Decomposição Funcional

  Nesta abordagem, o foco está no tipo de computação realizada, em vez de nos dados manipulados pela computação. O problema é decomposto de acordo com o trabalho a ser realizado. Cada tarefa então realiza uma porção específica do trabalho global.

  Exemplo: previsão do clima, processamento de sinal.

Não há uma heurística pronta para fazer isso de maneira automática, e cabe ao programador levar em conta essas questões durante o processo criativo de paralelização da aplicação, utilizando, dentre outros, o método de tentativa e erro para obter o melhor desempenho possível.

2.3 Balanceamento de carga

O balanceamento de carga é muito importante para o desempenho das aplicações paralelas, já que assegurar uma parcela justa de atividade para cada processador pode representar um grande incremento no desempenho dessas aplicações. O objetivo é manter todos os processadores ocupados todo o tempo. Se em um dado momento algum processador estiver ocioso, isso evidencia que a carga de trabalho não está igualmente distribuída no sistema. Então possivelmente a aplicação não vai executar de modo tão rápido ou eficiente como poderia.

Os métodos de balanceamento de carga podem ser estáticos ou dinâmicos, como detalhados na próxima seção.

2.3.1 Balanceamento estático

O método mais simples de balanceamento de carga é o estático. Nesse método, o problema é dividido em tarefas e as tarefas são atribuídas a cada processador apenas uma vez.

O particionamento estático pode ser feito antes do início do trabalho, ou como o primeiro passo da aplicação. O tamanho e o número de tarefas pode ser ajustado dependendo da capacidade computacional de cada máquina.

Se a única informação utilizada para alocar o trabalho aos processadores for a dimensão dos dados, então o método é dito estático por natureza, porque não muda, independente do conteúdo dos dados, mas apenas se sua dimensão mudar.

Muitas operações de matrizes são facilmente decompostas em termos de subseções regulares da matriz original. Por exemplo, o cálculo do determinante usa apenas aspectos da estrutura da matriz e não o conteúdo dos dados de uma célula em particular. Nesse

caso, a computação pode ser facilmente dividida entre os processadores sem levar em conta os dados propriamente ditos, mas apenas o conhecimento acerca da dimensão da matriz.

De uma forma mais geral, se existem Math input error coisas para computar e Math input error processadores disponíveis para o trabalho, uma forma muito simples é atribuir Math input error para cada processador computar.

2.3.2 Balanceamento dinâmico

Por mais fáceis e intuitivos que os métodos estáticos possam ser, eles apenas focam na estrutura dos dados e não nas características dos dados em si mesmos. Por exemplo, se a matriz cujo determinante está sendo calculado for esparsa (ou seja, com maioria dos elementos iguais a 0), um método estático não levará isso em consideração. Já uma solução mais elaborada pode reconhecer essa informação como significativa e processar a matriz mais eficientemente.

Quando a carga computacional varia de acordo com os dados processados, métodos dinâmicos mais sofisticados de balanceamento de carga são necessários. Há vários métodos dinâmicos de balanceamento de carga, sendo que os mais relevantes são destacados a seguir.

Saco de tarefas

O "saco de tarefas" é um dos métodos mais comuns de balanceamento dinâmico de carga. Ele é tipicamente implementado como uma topologia mestre/trabalhador, onde o processo/*thread* mestre gerencia um conjunto de tarefas alocadas para os processos/*threads* trabalhadores. O mestre envia novas tarefas para os trabalhadores assim que eles ficam ociosos. A Figura 2.12 ilustra esse modelo de interação.

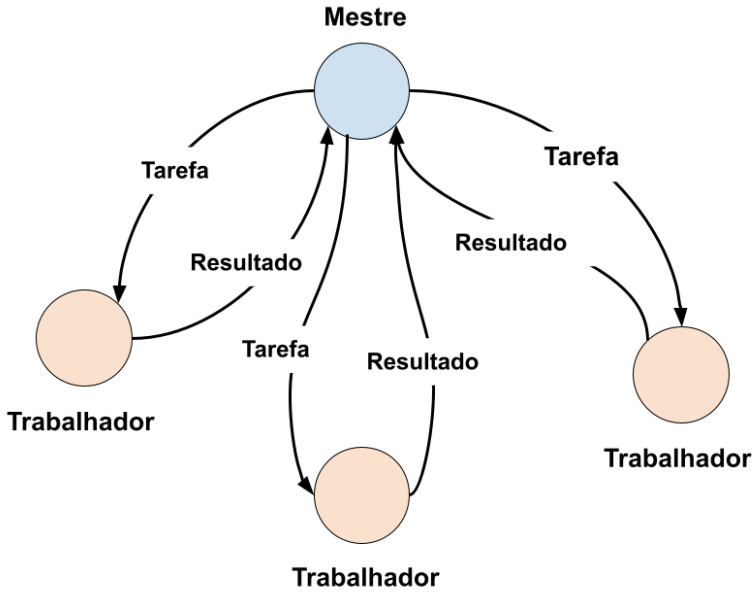

Figura 2.12: Modelo mestre/trabalhador usado para implementar o método saco de tarefas

O modelo mestre/trabalhador pode ser aplicado em diversas situações, sempre que não houver dependências de dados entre as tarefas da aplicação. Contudo, é importante destacar que esse modelo não costuma ser escalável para um grande número de processadores devido ao gargalo de comunicação com o nó mestre.

O modelo mestre/trabalhador também não é adequado para aplicações com comunicação de tarefa para tarefa, já que as tarefas iniciarão e terminarão em tempos arbitrários. Nesse caso, estratégias complementares precisam ser utilizadas. Por exemplo, em algum tempo predeterminado todos os processos param de executar; as cargas de trabalho são reexaminadas e redistribuídas se necessário. Outras variações desse modelo são possíveis para aplicações específicas.

Simulações *N-body*

Há uma classe de problemas de simulação computacional normalmente modelados como um conjunto de Math input error partículas unicamente identificadas (geralmente chamadas de

bodies), as quais podem ou não interagir umas com as outras, devendo ter alguns cálculos efetuados em cada uma delas.

Em alguns casos, especialmente naqueles onde não há forças de interação entre as partículas, essa classe de problemas pode ser reduzida para o caso de particionamento estático de dados. Mais frequentemente, contudo, cada partícula exerce alguma forma de influência sobre seus pares, algumas vezes de forma muito localizada (i.e., os efeitos têm curto alcance, como a força atômica forte), e outras vezes, não (como a gravidade).

Nos casos onde há áreas bem definidas de interação, normalmente é melhor atribuir todas as partículas interagindo entre si a um mesmo processador, e lidar com todas as iterações restantes (i.e., interações envolvendo partículas fora da área local) como se todas as partículas fossem apenas uma única grande partícula. O método de Barnes-Hut, apresentado na Seção 1.1.2, é um exemplo típico.

Método de refinamento adaptativo de malhas

Outra classe de problemas são os caracterizados pelo mapeamento dos dados em "grades" ou "malhas". Quanto mais estreita a malha, mais acurado o resultado, contudo, maior será o tempo de computação necessário. A Figura 2.13 [3] mostra essa ideia.

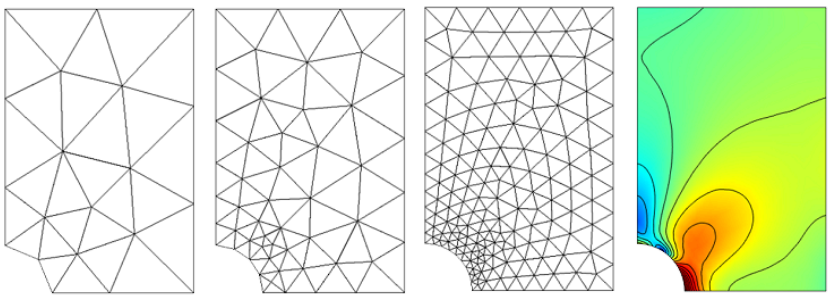

Figura 2.13: Método de refinamento adaptativo de malhas

Com frequência, não se sabe *a priori* quão fina deverá ser a malha até o início dos cálculos e se descubra, por exemplo, que o erro está

muito grande, requerendo uma malha mais fina e cálculos mais precisos. Um algoritmo sequencial apenas recalcularia os pontos da malha gerando automaticamente uma malha mais fina. Em uma aproximação paralela, o uso de métodos de refinamento adaptativo de malhas permite criar malhas mais finas ao longo da execução do programa e de acordo com a necessidade do problema. As malhas são compartilhadas pelos diferentes fluxos de execução e cada fluxo processa uma parte dela. Essa estratégia garante um melhor balanceamento de carga se comparado a uma divisão estática onde cada processador trabalha na sua parte inicial dos dados.

2.4 Avaliação de desempenho

O desempenho de um programa paralelo pode ser afetado por diversos fatores. O primeiro é a granularidade da tarefa, ou seja, a taxa entre o número de *bytes* recebidos e o número de operações de ponto flutuante efetuadas por um processo.

No caso de um sistema distribuído, o número total de mensagens enviadas também é um fator a ser considerado. Como uma regra geral, enviar um pequeno número de mensagens grandes levará menos tempo do que enviar um grande número de pequenas mensagens. Contudo, isso nem sempre ocorre, já que algumas aplicações podem sobrepor a computação com o envio de pequenas mensagens. O número ideal de mensagens é específico para cada aplicação.

Há aplicações bem adequadas ao paralelismo funcional, enquanto outras se aplicam bem ao paralelismo de dados (Seção 2.2.4). No paralelismo funcional diferentes máquinas fazem diferentes tarefas baseadas nas suas capacidades específicas. No paralelismo de dados, os dados são distribuídos para todas os processos ou *threads*. As operações (frequentemente bastante similares) são realizadas sobre cada conjunto de dados e a informação é

transmitida entre os processadores até a resolução final do problema.

Os programas paralelos podem também usar uma combinação de ambos os modelos para explorar totalmente as potencialidades de cada máquina. Computadores diferentes terão capacidades de processamento diferentes, e ainda que todas as máquinas sejam do mesmo tipo e modelo, elas podem apresentar diferenças de desempenho devido à carga de processamento proveniente dos outros usuários.

Considerações a respeito da rede de interconexão são também importantes, caso se esteja utilizando um *cluster* ou um computador similar. Os atrasos da comunicação na rede podem causar problemas. Além disso, a capacidade de processamento disponível pode variar dinamicamente dependendo da carga de trabalho de cada máquina.

Para lidar com esses problemas, alguma forma de balanceamento de carga (Seção 2.3) deve ser utilizada. Em qualquer dos casos, é necessário avaliar corretamente o desempenho de um programa. Algumas considerações sobre métricas de avaliação de desempenho são apresentadas na próxima seção.

2.4.1 Medidas de desempenho

Uma das questões que surgem ao elaborar um programa paralelo é saber se o seu desempenho é adequado quando executado em um ambiente paralelo. Deste modo, para avaliar o desempenho de um sistema de processamento paralelo, as métricas mais importantes para serem consideradas são: *speedup* (também conhecido como ganho de desempenho ou aceleração), eficiência e escalabilidade.

Diversos fatores influenciam essas métricas, tais como o custo de sincronização e comunicação, a distribuição das tarefas entre os processadores e o percentual do tempo de execução do programa

passível de paralelização. Com relação a este último item, alguns estudos teóricos auxiliam na previsão do ganho de desempenho máximo a ser obtido com a paralelização de uma aplicação sequencial. São estes: a *Lei de Amdahl* e a *Lei de Gustafson*.

Speedup

O ***speedup***, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador (Math input error) e o tempo gasto na execução com Math input error processadores (Math input error), como visto na equação a seguir:

Math input error

Em condições ideais, quando o *speedup* é sempre igual a Math input error, onde Math input error é o número de processadores em uso, temos o chamado *speedup* linear. Mas, em geral, o *speedup* é menor do que Math input error, devido principalmente à sobrecarga de comunicação entre os diferentes fluxos de execução do programa, perdas por sincronização e decomposição de tarefas mal feita. Quando isso acontece, chamamos o *speedup* de sublinear. Essa situação pode se deteriorar até o ponto no qual a adição de mais processadores diminui o ganho obtido, em vez de aumentar, caracterizando um "retorno negativo".

Em algumas situações especiais, *speedups* superiores a Math input error podem ser obtidos (denominados "*speedups* superlineares"). Exemplos destas situações são aplicações onde o volume de dados manipulados é grande o suficiente para exceder o tamanho da memória cache de um único processador. Nesse caso, ao dividir a aplicação entre Math input error processadores, o volume de dados manipulado por cada processador é aproximadamente dividido por Math input error, sendo este volume agora pequeno o suficiente para poder ser armazenado integralmente, ou com baixo nível de interferência destrutiva, na

memória cache de cada processador. Dentro deste quadro, e respeitadas as condições mencionadas anteriormente, é razoável se esperar um *speedup* superior a Math input error no processo de paralelização, já que o desempenho com um único processador fica muito prejudicado pela baixa taxa de acerto nos acessos à memória cache.

Uma situação análoga ocorre quando são feitas buscas em grandes bases de dados, tais como a busca de dados genômicos realizadas por diversas implementações paralela do programa BLAST. Nesse caso, a quantidade de memória RAM acumulada de cada um dos nós do *cluster* permite a movimentação da base de dados do disco inteiramente para a memória. Portanto, isso reduz dramaticamente o tempo requerido, por exemplo, para o mpiBLAST percorrer toda a base de dados (Correa and Silva 2012).

O *speedup* superlinear pode ocorrer também, sob determinadas condições, quando da execução de algoritmos de *backtracking* e *branch and bound* paralelos (Rao and Kumar 1993; Quinn and Deo 1986). A Figura 2.14 ilustra diferentes curvas de *speedups*.

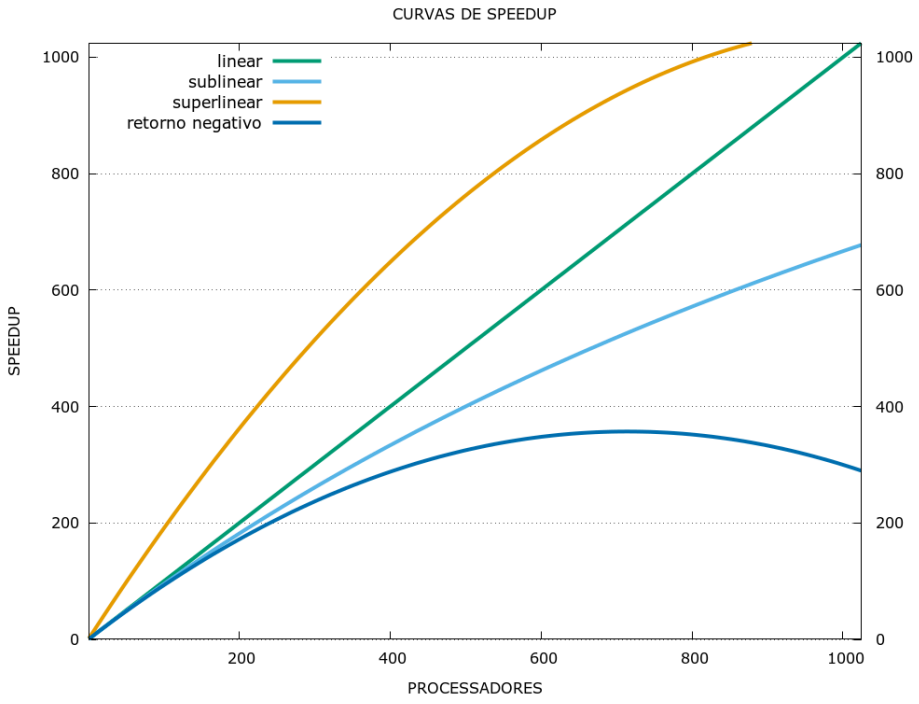

Figura 2.14: Curvas de *speedup*

Outra situação onde *speedups* superiores a Math input error podem ser produzidos se dá quando a complexidade do problema a ser resolvido é Math input error, onde Math input error. Nestes casos, ao se distribuir de forma homogênea os dados da aplicação entre os Math input error processadores no processo de paralelização, o tempo gasto por cada processador para resolver o problema sobre este subconjunto de dados, respeitadas as condições mencionadas anteriormente, é aproximadamente:

Math input error

onde Math input error é o tempo gasto para se obter a solução do problema com um único processador. Portanto, um *speedup* superlinear é obtido. Esse tipo particular de situação pode ser observado, por exemplo, em aplicações para ordenação de um vetor de números.

Eficiência

A **eficiência** é uma medida de quão efetiva é a adição de novos processadores para ajudar na resolução de um problema. Seu valor é obtido pela razão entre o *speedup* (Math input error) e o número de processadores (Math input error) utilizados para obter este *speedup*, conforme podemos observar na seguinte equação:

Math input error

Novamente, como o *speedup* em geral é menor do que Math input error por conta da sobrecarga do processamento paralelo, a eficiência tipicamente assume um valor menor do que 1, mas, preferencialmente, próximo de 1. Para se obter *speedup* muito próximo de Math input error e, consequentemente, uma eficiência

muito próxima de 1, as seguintes condições devem normalmente ser satisfeitas:

- no código a ser paralelizado, o percentual de código não paralelizável (que continuará sendo executado de forma sequencial) é muito pequeno;
- no processo de paralelização, a distribuição de carga de trabalho entre os Math input error processadores deve ser homogênea;
- os processadores trabalham nos trechos de código executados em paralelo de forma bastante independente, requerendo muito pouca comunicação ou sincronização entre eles.

Escalabilidade

Um sistema é dito escalável quando sua eficiência se mantém constante à medida que o número de processadores Math input error aplicado à solução do problema cresce. Se o tamanho do problema é mantido constante e o número de processadores aumenta, o *overhead* de comunicação tende a crescer e a eficiência, a diminuir. Na prática, uma análise da escalabilidade deve considerar a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que Math input error cresce de forma a contrabalançar o natural aumento do *overhead* de comunicação quando Math input error cresce.

Considere como exemplo um problema de tamanho Math input error . Usando Math input error processadores, esse problema leva um tempo Math input error para ser executado. O sistema é dito escalável se um problema de tamanho Math input error executado em Math input error processadores leva o mesmo tempo Math input error. Escalabilidade é frequentemente uma propriedade

mais desejável que o *speedup*. Veja maiores detalhes na Seção 2.4.3.

2.4.2 Lei de Amdahl

A lei de Amdahl[4] (Amdahl 1967) é usada para determinar o *speedup* teórico máximo de uma aplicação, o qual está limitado pelo tempo necessário para executar a fração sequencial desse programa (a parte do programa não paralelizada). Considerando que a fração de tempo sequencial do programa é conhecida e dada por Math input error e que o programa paralelo será executado em Math input error processadores, então a lei de Amdahl define o *speedup* teórico (Math input error) máximo conforme determinado na equação final a seguir:

Math input error

Math input error

Math input error

A Figura 2.15 ilustra sua definição. Podemos verificar que, mesmo com um número infinito de processadores, o *speedup* máximo está limitado a:

Math input error

onde Math input error é a fração sequencial do programa.

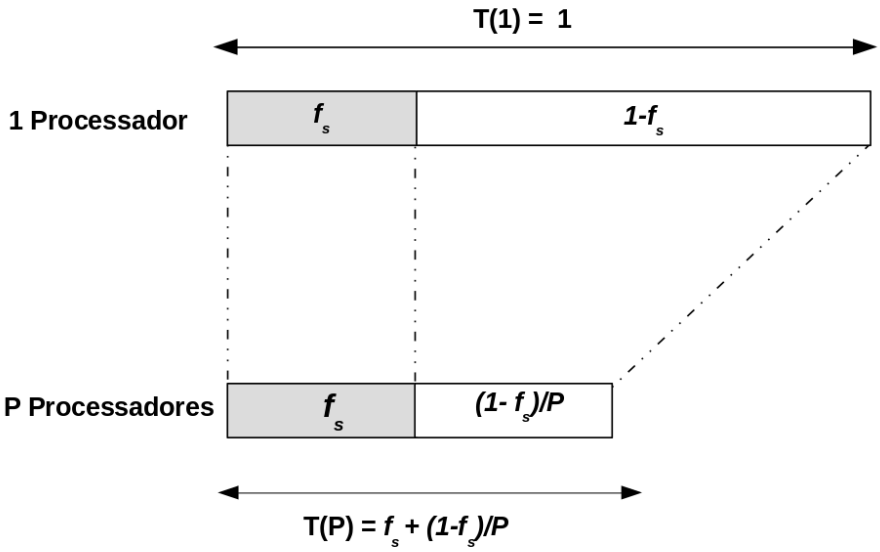

Figura 2.15: Lei de Amdahl

Considere, por exemplo, um programa que, por hipótese, demora um total de 20 horas quando executado em um único processador, sendo que existe uma fração não paralelizável que leva 1 hora para ser executada. Então, por mais processadores que sejam utilizados, ao se reduzir a fração paralelizável a um valor perto de zero, o *speedup* máximo estará limitado a 20:

Math input error

A Figura 2.16 [5] apresenta exemplos de limites teóricos para o *speedup* com base na fração não paralelizável do programa. Com apenas 5% da computação sendo sequencial, observamos um *speedup* teórico máximo de 20, não importando o número de processadores em uso. Se 20% da computação for sequencial, o *speedup* máximo será de 5.

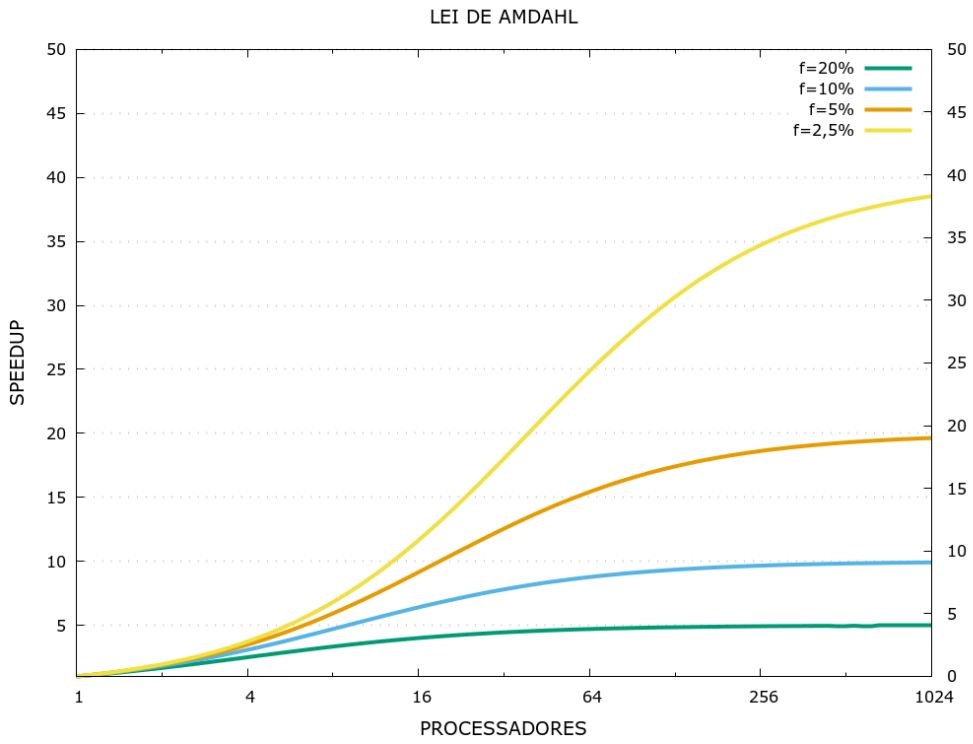

Figura 2.16: Limite teórico do *speedup* segundo a Lei de Amdahl

2.4.3 Lei de Gustafson

A lei de Gustafson (Gustafson 1988), também conhecida como lei de Gustafson–Barsis, afirma que a computação envolvendo um conjunto arbitrariamente grande de dados de entrada pode ser eficientemente paralelizada, oferecendo um contraponto à lei de Amdahl, que toma como base a premissa de um problema de tamanho fixo, independentemente do número de processadores utilizados.

A proposta de Gustafson considera que os programadores tendem a aumentar o tamanho dos problemas à medida que o número de processadores aumenta, de modo a manter fixo o tempo de resolução de um problema. Assim, à medida que mais processadores estão disponíveis, problemas maiores podem ser resolvidos

consumindo a mesma quantidade de tempo, resultando em uma diminuição proporcional da fração sequencial.

Nos dois casos, tanto na Lei de Amdahl como na de Gustafson, assumimos uma distribuição igual da carga de processamento entre todos os processadores.

Segundo Gustafson, o tempo de execução de um programa em um computador paralelo pode ser definido como a soma de duas parcelas:

Math input error

onde Math input error é o tempo sequencial (Math input error) e Math input error é o tempo paralelo (Math input error), em qualquer um dos processadores Math input error, sem considerar o *overhead*, como pode ser visto na Figura 2.17. Considerando que Math input error, o tempo paralelo por processador, deve ser fixado enquanto Math input error é variado, o tempo correspondente para o processamento sequencial é dado por:

Math input error

O *speedup* é definido, correspondentemente, como:

Math input error

Teremos, por fim:

Math input error

O que nos leva à forma final da lei de Gustafson, resumida na equação seguinte:

Math input error

onde Math input error é o número de processadores, Math input error é o *speedup* teórico e Math input error é a maior fração não paralelizável do programa.

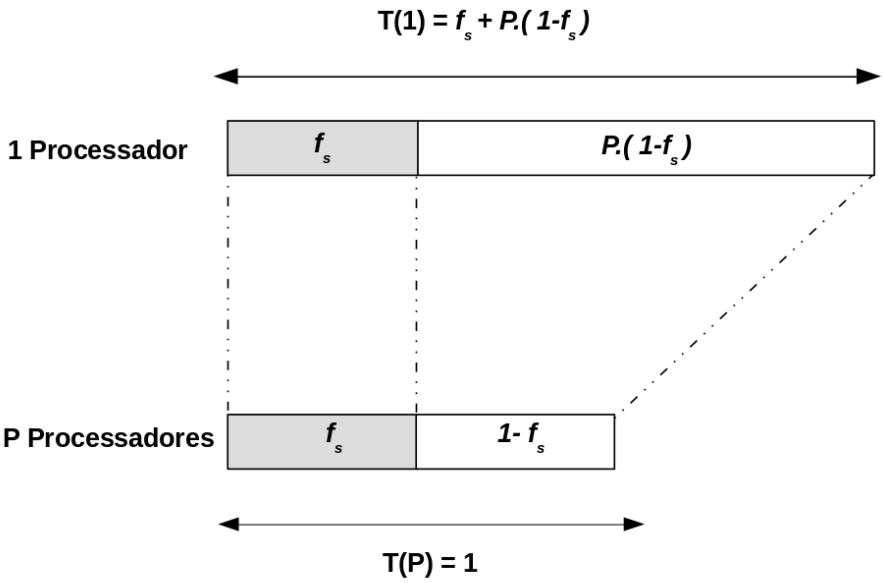

Figura 2.17: Lei de Gustafson

Então, se Math input error é pequeno, o *speedup* é aproximadamente Math input error, como esperado. Pode até mesmo acontecer que Math input error diminua à medida que Math input error (junto com o tamanho do problema) aumenta; se isso se verifica, então Math input error se aproxima monotonicamente de Math input error com o crescimento de Math input error.

Por exemplo, se um programa possui uma parte sequencial de tempo constante equivalente a 5% do tempo total da execução paralela, conforme aumenta-se a quantidade de processadores (e o tamanho do problema), o crescimento do *speedup* se mantém. Isso significa que, neste exemplo, considerando uma execução com 2 processadores, o *speedup* calculado é de 1,95; para 20 processadores, o novo *speedup* calculado é de 19,05; e se aumentar

ainda mais a quantidade de processadores, para 200, o *speedup* é de 190,05. Esses e outros valores podem ser visto no gráfico da Figura 2.18.

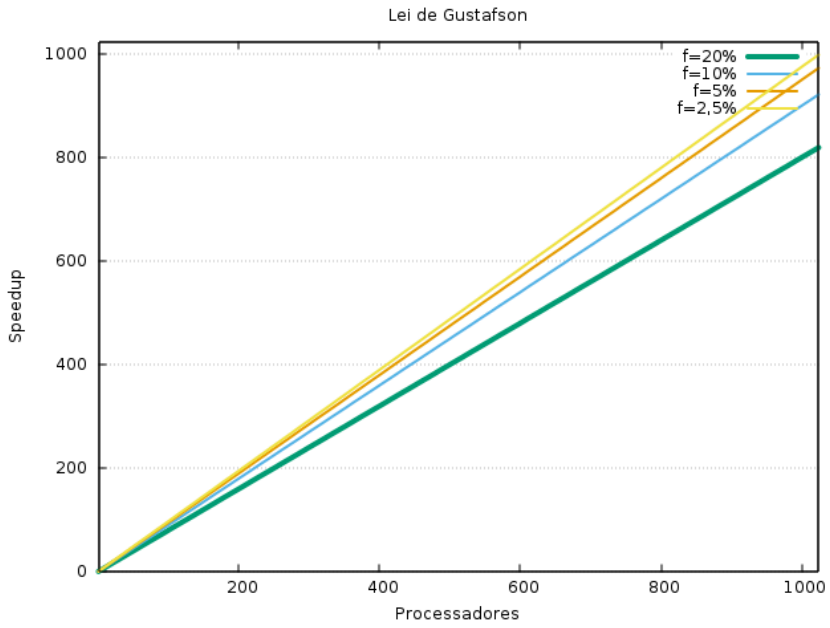

Figura 2.18: Limite teórico do *speedup* segundo a Lei de Gustafson

Na prática, o resultado final do *speedup* deverá ser algum valor entre a proposta otimista de Gustafson e a pessimista de Amdahl, desde que o programa tenha sido adequadamente elaborado, com balanceamento adequado de carga e baixa carga de sincronização, segundo os métodos já descritos neste capítulo.

2.5 Arquiteturas paralelas

As arquiteturas paralelas são aquelas capazes de explorar o paralelismo existente nas aplicações e podem ser organizadas de diversas maneiras. Vamos apresentar, para melhor entendimento dos paradigmas de programação ilustrados no livro, algumas das formas de organizar essas arquiteturas paralelas.

Ao falarmos sobre paralelismo, devemos considerar os seguintes níveis de paralelismo possíveis na execução de uma aplicação:

- no nível de instrução (granulosidade fina): encontrado em processadores com *pipeline*, superescalares ou VLIW (*Very Long Instruction Word*). Esse tipo de paralelismo é explorado de forma transparente ao usuário, normalmente pelo compilador ou diretamente pelo *hardware*, pela execução de diversas instruções em paralelo em diversas unidades funcionais existentes em um processador.

- no nível de *thread* (granulosidade média): encontrado em diversos tipos de processadores e arquiteturas, destacando-se as *multithreading*, de *multithreading* simultâneo (SMT), *multicore* e aceleradores. Nesse caso, é necessário que tenhamos várias *threads* executando em paralelo, necessitando da intervenção do programador ou do compilador para o aproveitamento desse tipo de paralelismo.

- nível de processo (granulosidade grossa): encontrado em multiprocessadores (memória compartilhada) e multicomputadores (memória distribuída), e requer a também a intervenção explícita do programador e o suporte de bibliotecas e/ou ambientes de execução paralelos.

As arquiteturas que exploram o paralelismo no nível de instrução não vão receber nossa atenção, pois a exploração do paralelismo é feita diretamente pelo *hardware* ou compilador e não necessitam da intervenção do programador.

Dentre as arquiteturas que exploram o paralelismo no nível de *thread*, temos dois tipos de arquiteturas que vamos estudar mais detalhadamente: as *multicore* e os aceleradores.

As arquiteturas *multicore*, também conhecidas como *chip multiprocessor* (CMP), são caracterizadas pela existência de diversos processadores (núcleos ou *cores*) em um mesmo encapsulamento,

compartilhando uma memória cache e a memória principal. Eventualmente esse tipo de arquitetura pode ser expandido para vários *chips*, cada um com vários núcleos, compartilhando uma mesma memória global, no que é chamado multiprocessamento simétrico (SMP, do inglês *Symmetric Multiprocessing*). Note que, em ambos os casos, a comunicação entre as *threads* é feita através de variáveis na memória compartilhada (ou na memória cache no caso dos *multicore*), necessitando de intervenção explícita do programador para a coordenação do paralelismo. Neste livro, apresentaremos o OpenMP para a exploração desse tipo de paralelismo.

As arquiteturas com aceleradores são um tipo particular de exploração de paralelismo no nível de *thread*, onde trechos computacionalmente intensivos do programa, chamados de *kernels*, são enviados para execução nos aceleradores, que têm memória distinta da memória do hospedeiro. Isso requer que tanto o código como os dados sejam transferidos da memória do hospedeiro para a memória do acelerador, através de barramentos dedicados de alta velocidade. Os tipos de aceleradores mais utilizados nesse tipo de paralelismo são as GPUs (Graphics Processing Unit), cujos detalhes vamos apresentar mais adiante, junto da biblioteca de programação OpenACC.

As **arquiteturas de memória compartilhada** têm como característica principal diversos processadores compartilhando um único espaço de endereçamento, permitindo assim a comunicação entre os diferentes fluxos de execução, sejam processos ou *threads*, por meio de variáveis na memória compartilhada. Contudo, o paralelismo no nível de processo em multiprocessadores com memória compartilhada, incluídos aí os CMPs e SMPs, não é usual. A comunicação via memória entre processos, que têm espaços de endereçamento distintos, requer esquemas mais sofisticados e lentos de comunicação via páginas compartilhadas pelo sistema operacional. Nesse caso é preferível a utilização de *threads* e, dentro

os diversos tipos de biblioteca disponíveis para exploração desse tipo de paralelismo, vamos apresentar neste livro o uso do OpenMP.

Já nas **arquiteturas com memória distribuída**, ou multicomputadores, cada processador possui um espaço de endereçamento próprio que não é compartilhado com os demais processadores. Desta forma, a troca de informações entre os diversos fluxos de execução deve ser feita por meio de troca de mensagens transmitidas através de uma rede de interconexão ligando esses processadores, caracterizando o que também é chamado de sistema distribuído. O paralelismo é explorado no nível de processos, que se comunicam por rotinas que realizam a troca de mensagens pela rede, como o MPI, que vamos estudar neste livro.

Um exemplo típico dessa categoria de computadores são os *clusters*, ou seja, um tipo de sistema de processamento paralelo onde uma coleção de computadores independentes se interconectam através de uma rede de comunicação, trabalhando cooperativamente como um único e integrado recurso computacional.

Antes de apresentarmos maiores detalhes de algumas dessas arquiteturas, vamos introduzir a Classificação de Flynn (Flynn 1966), que considera os seguintes tipos de fluxos possíveis em um programa em execução:

- Single Instruction Stream (SI)
- Multiple Instruction Streams(MI)
- Single Data Stream (SD)
- Multiple Data Streams (MD)

A associação desses diversos tipos de fluxos dá origem às seguintes categorias de organização de um computador:

- SISD – Single Instruction Stream, Single Data Stream – (Exemplos: processadores convencionais, *pipelined* ou superescalares)

- SIMD – Single Instruction Stream, Multiple Data Streams – (Exemplos: processadores vetoriais, unidades funcionais vetoriais, arquiteturas SIMD)

- MIMD – Multiple Instruction Streams, Multiple Data Streams – (Exemplos: multiprocessadores, multicomputadores)

Vejamos alguns detalhes dessas arquiteturas a seguir.

2.5.1 Arquiteturas SISD

São os processadores convencionais, onde apenas um processo ou *thread* é executado por vez, podendo fazer uso de técnicas de exploração de paralelismo temporal (*pipeline*) ou espacial (superescalares) em *hardware*.

Nesse caso, o paralelismo, se explorado, é feito de forma transparente ao usuário, no nível de instrução, com um suporte mínimo do compilador realizando, por exemplo, o escalonamento mais adequado das instruções em linguagem de máquina para a execução em várias unidades funcionais.

2.5.2 Arquiteturas SIMD

As arquiteturas SIMD consistem basicamente de uma única instrução operando sobre diversos dados diferentes, como ocorre, por exemplo, nos processadores vetoriais, cujo caso mais famoso são os computadores da linha Cray (Wikipedia 2020a).

- Processadores Vetoriais
- Arquiteturas SIMD convencionais
- Arquiteturas Sistólicas

Processadores Vetoriais

Os processadores vetoriais são arquiteturas *pipelined* do tipo SIMD, ou seja, uma única instrução opera sobre vários dados. Um processador vetorial possui instruções para operar em vetores: um conjunto linear de valores. Uma operação vetorial típica pode adicionar dois vetores com *N* elementos em notação ponto flutuante para obter um único vetor de resultado com *N* elementos.

A instrução vetorial é equivalente a um laço inteiro, onde cada iteração computa vários elementos do resultado, atualiza os índices e retorna para o início. As instruções vetoriais possuem as seguintes características:

- Cada instrução equivale a um laço;
- O cálculo de cada resultado não depende de resultados anteriores, assim é possível haver pipelines profundos sem a ocorrência de dependências de dados; e
- O padrão de acesso à memória para a busca dos operandos é conhecido e regular, beneficiando-se da utilização de memória com entrelaçamento (*interleaving*) ou mesmo memórias caches.

Arquiteturas SIMD convencionais

É uma classe importante de processadores, devido a fatores como:

- Simplicidade de conceitos e programação;
- Regularidade da estrutura;
- Facilidade de escalabilidade em tamanho e desempenho; e
- Aplicação direta em uma série de aplicações paralelas para obter o desempenho necessário.

Os processadores executam sincronizadamente a mesma instrução sobre dados diferentes. Utiliza vários processadores especiais muito

mais simples, organizados em geral de forma matricial. É muito eficiente em aplicações onde cada processador pode ser associado a uma submatriz independente de dados (processamento de imagens, algoritmos matemáticos etc.).

Arquiteturas Sistólicas

Arquiteturas sistólicas foi um termo utilizado por H.T. Kung, em 1978 para definir um tipo de arquitetura na qual os dados são "bombeados" através de um estrutura em *hardware*, em analogia com o funcionamento do coração.

Uma arquitetura sistólica consiste em um conjunto de células interconectadas, cada uma delas capaz de realizar uma operação simples. O uso de estruturas de comunicação e controle, simples e regulares, oferecem uma grande vantagem quando comparado com as estruturas mais complexas das arquiteturas convencionais. (Kung 1982)

2.5.3 Arquiteturas MIMD

As arquiteturas MIMD se subdividem em dois grandes grupos:

- Arquiteturas de Memória Distribuída – Cada processador é capaz de endereçar apenas a sua própria memória local.
- Arquiteturas de Memória Compartilhada – Cada processador consegue ter acesso a um espaço de endereçamento global do sistema.

MIMD com Memória Distribuída

Nas arquiteturas MIMD cada processador enxerga apenas o seu espaço de memória, sem ter acesso direto à memória utilizada pelos outros processadores do sistema. Essas arquiteturas apresentam

como vantagens o fato de serem altamente escaláveis e permitirem a construção de processadores maciçamente paralelos, com milhares ou mesmo milhões de processadores. Dessa forma, a comunicação entre os processadores é feita através de troca de mensagens por meio de uma rede de comunicação que interliga os diversos computadores, como pode ser observado na Figura 2.19. Note que a troca de mensagens resolve tanto o problema da comunicação dos processadores como o da sincronização entre os diversos processos em execução.

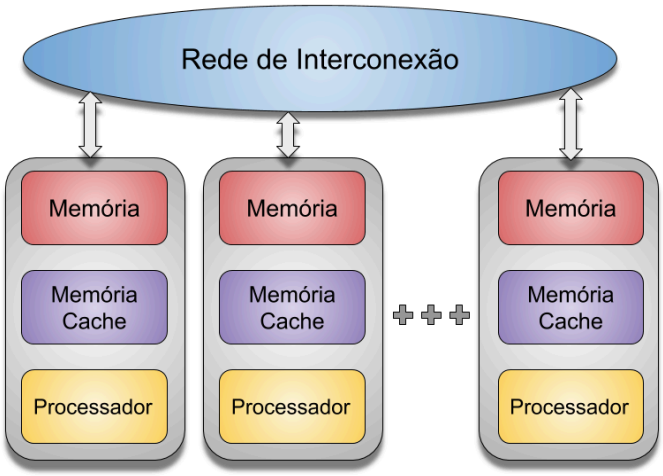

Figura 2.19: Arquiteturas MIMD com Memória Distribuída

As suas grandes desvantagens são a necessidade de se fazer uma boa distribuição de carga entre os processadores, quer seja automaticamente, quer seja manualmente. É necessário ainda evitar as situações de *deadlock*, tanto no nível de aplicação como no nível de sistema operacional, que podem ocorrer quando há uma espera circular pelo envio e recebimento de mensagens entre os processos. Além disso, é um modelo de programação menos natural, demandando diversas modificações em relação ao código convencional sequencial.

MIMD com Memória Compartilhada

Nessas arquiteturas, cada processador tem acesso a um espaço de endereçamento global comum a todos processadores, permitindo que técnicas de programação sequenciais sejam facilmente adaptadas, já que não há necessidade do particionamento de código ou dados. Também não há necessidade da movimentação física dos dados, quando dois ou mais processadores se comunicam, resultando em uma comunicação entre processos ou *threads* bastante eficiente. A Figura 2.20 mostra um exemplo dessa arquitetura.

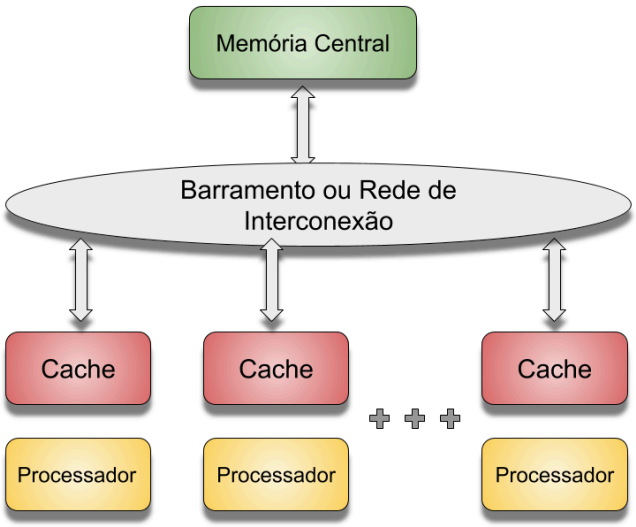

Figura 2.20: Arquiteturas MIMD com Memória Compartilhada

Entre suas desvantagens podemos destacar a obrigatoriedade do uso de primitivas especiais de sincronização, quando do acesso a regiões compartilhadas de memória, para assegurar um resultado correto para a computação. Há também a falta de escalabilidade devido ao problema de contenção de memória, já que depois de um determinado número de processadores, a adição de mais processadores não resulta em nenhuma melhoria no desempenho.

Essas arquiteturas possuem diversas classificações e variações em sua construção, das quais destacamos algumas a seguir.

Arquiteturas UMA

São arquiteturas com memória única global, onde o tempo de acesso à memória é uniforme para todos os nós de processamento e, normalmente, o nós de processamento e a memória são interconectados através de barramento único. Possuem um número reduzido de nós de processamento e a coerência das caches é mantida por *hardware* com o uso da técnica de *snooping*. Essas arquiteturas são conhecidas também como *multicore*, multiprocessador em chip (CMP), ou multiprocessamento simétrico (SMP).

Um dos grandes problemas dos sistemas de memória compartilhada com uso de barramento único é a escalabilidade. Ou seja, para um grande número de processadores a contenção no uso do barramento aumenta até o ponto no qual a adição de novos processadores não é mais efetiva. Um barramento único tem um tamanho (comprimento físico) limitado, diminuindo o número máximo de processadores e módulos de memória que podem ser acrescidos ao sistema.

Uma estratégia comum utilizada nos modernos processadores comerciais é agregar um ou mais níveis de cache a cada um dos processadores (núcleos), incluindo-se aí uma cache compartilhada de nível 3, utilizada para a comunicação entre os núcleos, sem que haja necessidade de acesso ao barramento e/ou memória principal. O uso de caches, por outro lado, exige o uso de algoritmos em *hardware* para manutenção da coerência de dados entre as diversas caches.

Em vez do barramento, pode-se utilizar uma rede de interconexão com alto *throughput* e baixa latência. Essa rede tem características distintas do barramento, pois permite o acesso simultâneo de vários processadores a módulos de memória distintos. Uma solução utilizada para permitir que a quantidade de memória do sistema cresça à medida que cresce o número de processadores é agregar um módulo de memória local para cada processador do sistema.

Esses sistemas são chamados de MIMD com memória compartilhada distribuída, ou ainda, arquiteturas com tempo de acesso não uniforme à memória (*Non-uniform Memory Access* – NUMA), conforme veremos a seguir.

Arquiteturas NUMA

Nessas arquiteturas, a memória é dividida em tantos blocos quanto forem os processadores do sistema, e cada bloco de memória é conectado via barramento a um processador com memória local. O acesso aos dados na memória local é muito mais rápido do que o acesso aos dados em blocos de memória remotos e essa diferença faz com que sejam necessários cuidados especiais ao se programar em arquiteturas desse tipo.

Um dos problemas decorrentes dessa estratégia é a necessidade de se tolerar e esconder as latências de acesso aos dados nas memórias mais distantes. Uma rede de interconexão com alto desempenho deve ser utilizada para reduzir o tempo de acesso aos módulos de memória remotos, melhorando a escalabilidade. A busca avançada dos dados/páginas é uma outra solução empregada para diminuir a latência.

O uso de memórias caches locais, com cópia dos conteúdos das memórias remotas, também reduz o problema da contenção e da latência, pois não é necessário atravessar a rede de interconexão, mas isso exige também o uso de esquemas complexos para a manutenção da consistência de dados nas várias caches locais.

Apesar do uso de memória compartilhada, as arquiteturas NUMA mais modernas também oferecem bibliotecas para programação utilizando troca de mensagens.

2.5.4 Aceleradores

Uma opção importante para obter um alto grau de paralelismo com baixo consumo de energia são os aceleradores. Como já dito, aceleradores são dispositivos especiais de *hardware* que trabalham em conjunto com processadores convencionais, executando trechos intensivos de código, com alto potencial de paralelismo, chamados de *kernels*. Entre os dispositivos desse tipo podemos destacar: processadores *manycores*; *Graphics Processing Units* (GPUs) e *Field Programmable Gate Arrays* (FPGAs) (Figura 2.21).

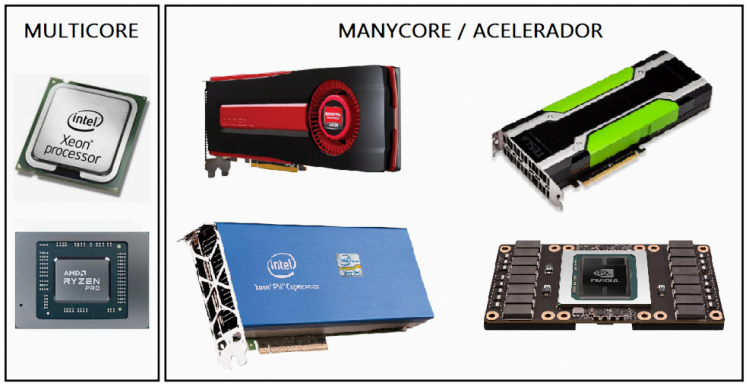

Figura 2.21: Tipos de arquiteturas: multicore, GPU e manycore

Os *Field Programmable Gate Arrays* (FPGAs) são dispositivos eletrônicos programáveis por *software*, que podem ser personalizados para executar com eficiência determinados trechos de código com uso intensivo de dados e/ou alta demanda computacional. Assim, esses dispositivos podem ser programados e reprogramados de acordo com o tipo de aplicação executada, se adaptando assim à solução de diversos tipos de problema.

Os processadores *manycore* são projetados para um alto grau de processamento paralelo, contendo vários núcleos de processadores simples e independentes (variando de algumas dezenas a centenas de núcleos). Os processadores *manycore* são amplamente utilizados em sistemas embarcados e em computação de alto desempenho (Wikipedia 2020b).

Em 2010 a Intel iniciou os primeiros estudos para o desenvolvimento da arquitetura MIC (*Many Integrated Core*). A arquitetura do Intel MIC oferece paralelismo maciço e vetorização com foco em computação de alto desempenho, demandando grandes volumes de dados para processamento. Aplicações de alto desempenho estão presentes em todos os setores, como a engenharia, medicina, economia e finanças (Rahman 2013).

A GPU (*Graphical Processing Unit*), desenvolvida pela NVIDIA nos anos 90, se baseia em um grande número de núcleos para processamento paralelo maciço com foco na eficiência energética e para aplicações com demandas que melhorem o *throughput*. Inicialmente desenvolvidos com o objetivo de atender a área de jogos, rapidamente mostrou-se muito eficiente na execução de vários tipos de aplicação científica (Zanotto et al. 2012).

Arquitetura dos Aceleradores Gráficos

As arquiteturas dos aceleradores gráficos (GPUs) são bem diferenciadas das arquiteturas dos processadores convencionais. O paralelismo nos aceleradores gráficos é explorado através de um conjunto maciço de multiprocessadores de fluxo (*streaming multiprocessors (SM)*), executando em paralelo e de forma sincronizada trechos computacionalmente intensivos das aplicações.

Para o melhor entendimento dos aceleradores gráficos (GPUs) vamos estudar, sem perda de generalidade, a arquitetura de um tipo de acelerador gráfico desenvolvido pela NVIDIA, a arquitetura Kepler.

Na Figura 2.22 verificamos que o acelerador gráfico possui uma arquitetura distinta, com diversos níveis de hierarquia de memória, algumas delas compartilhadas, outras exclusivas de cada multiprocessador de fluxo (SM). Analisamos esses e outros detalhes a seguir.

Figura 2.22: Arquitetura NVIDIA Kepler

Principais características da arquitetura Kepler

Cada unidade de multiprocessador de fluxo possui 192 núcleos CUDA de precisão simples e 64 de precisão dupla, onde cada núcleo tem unidades lógicas de aritmética inteira e de ponto flutuante capazes de operar em modo "pipeline", incluindo operações do tipo *fused multiply-add* (FMA). As 32 unidades de função especial (SFU) dentro de cada SM são utilizadas para aproximar operações transcendentais como raiz quadrada, seno, cosseno e recíproco (1/x). O projeto dessa arquitetura está focado no desempenho/consumo energético, fundamental na computação de alto desempenho moderna, e pode ser visto na Figura 2.23.

Figura 2.23: Multiprocessador de fluxo (SM)

O escalonador do multiprocessador de fluxo (SM) dispara as *threads* em grupos de 32 *threads* chamadas de *warps*. Cada SM possui quatro escalonadores de *warp*, permitindo um máximo de quatro *warps* disparadas e executadas concorrentemente. O número de registradores pode chegar até 255 registradores utilizados simultaneamente por cada *thread*.

Para melhorar ainda mais o desempenho, a arquitetura Kepler apresenta uma instrução de *shuffle* que permite às *threads* dentro de uma mesma *warp* compartilhar dados. Anteriormente, o compartilhamento de dados entre *threads* demandava o acesso à memória compartilhada, com operações de *load* e *store*, impactando em muito o desempenho de aplicações como, por exemplo, a transformada de Fourier (FFT).

Outro tipo de instrução disponível são operações atômicas em memória, permitindo que as *threads* realizem adequadamente operações de *read-modify-write*, como soma, máximo, mínimo e compare-e-troque em estruturas de dados compartilhadas. Operações atômicas são amplamente utilizadas para ordenação em paralelo e para o acesso em paralelo a estruturas de dados compartilhadas sem a necessidade de travas que serializam a execução do código.

A arquitetura de memória do acelerador está organizada em diversos níveis, possuindo uma cache L1 para cada SM, além de uma cache apenas de leitura, como visto na Figura 2.24.

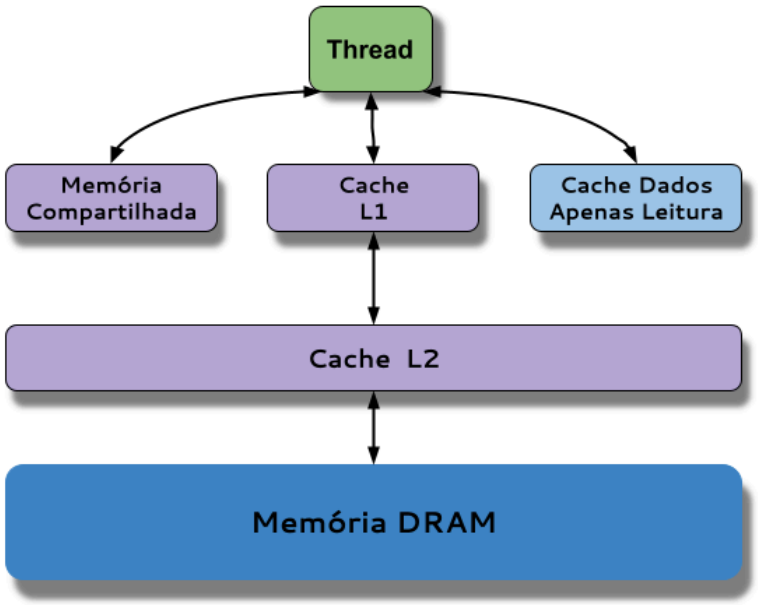

Figura 2.24: Hierarquia de memória

A quantidade de memória de cada SM é configurável, por exemplo, a memória local (64 ou 128 KB) pode ser dividida nas seguintes proporções: 75% x 25%, 25% x 75% ou 50% x 50% entre uma memória compartilhada e uma cache L1.

Além da cache L1, a arquitetura Kepler introduz uma cache apenas de leitura de 48KB. O gerenciamento dessa cache pode ser feito automaticamente pelo compilador ou explicitamente pelo

programador. O acesso a uma variável ou estrutura de dados que o programador identifica como apenas de leitura, pode ser declarada com a palavra-chave *const* *__restrict*, permitindo ao compilador carregá-la na cache apenas de leitura.

Essa arquitetura possui também um cache de nível 2 (L2) com 1,5 MB de capacidade. A cache L2 é o ponto primário de unificação de dados entre os diversos SMs, servindo operações de *load*, *store* e de textura, provendo um compartilhamento de dados eficiente e de alta velocidade.

Algoritmos onde o endereço dos dados é conhecido previamente, tais como solucionadores de física, *ray tracing* e multiplicação esparsa de matrizes, se beneficiam especialmente das hierarquia de cache. Os *kernels* de filtro e convolução onde é necessário que diversos SMs leiam os mesmos dados, também se beneficiam dessa hierarquia.

A arquitetura possui diversas outras facilidades como código de correção de erro, paralelismo dinâmico, gerenciamento de filas de trabalho e unidade de gerenciamento de *grids*, que servem para melhorar o desempenho e a confiabilidade do acelerador. Maiores detalhes podem ser vistos na referência (NVIDIA 2014). A arquitetura dos aceleradores está em constante evolução. Na Tabela 2.2 apresentamos as características de alguns aceleradores NVIDIA lançados mais recentemente.

Tabela 2.2: Comparação entre arquiteturas Nvidia

Características da GPU	NVIDIA Kepler	NVIDIA Tesla P100	NVIDIA A100
Versão	GK110	GP100	GA100
Arquitetura GPU	Kepler	Pascal	Ampere
Capac. Comput. (CC)	3.5	6.0	8.0
Threads / Warp	32	32	32
Máx. Warps / SM	64	64	64

Máx. Threads / SM	2048	2048	2048
Máx. Blocos Thread / SM	16	32	32
Máx. Reg. 32-bit / SM	65536	65536	65536
Máx. Reg. / Block	65536	65536	65536
Máx. Reg. / Thread	255	255	255
Máx. Tam. Bloco Threads	1024	1024	1024
Núcleos FP32 / SM	192	64	64
Reg. SM / Núcleos FP32	341	1024	1024
Mem. Compart. / SM	até 48 KB	64 KB	até 48 KB

Para finalizar esta seção, gostaríamos de mostrar como se dá a execução das *threads* no acelerador, apresentando o modelo de programação paralela CUDA. O CUDA é uma combinação de plataforma de hardware/software possibilitando aos aceleradores gráficos executar programas escritos em C, C++, Fortran e outras linguagens.

Um programa CUDA invoca funções paralelas chamadas *kernels*, que são executados em várias *threads* paralelas. O programador ou compilador organiza essas *threads* em blocos de *thread* e grades de blocos de *thread*, conforme mostrado na Figura 2.25.

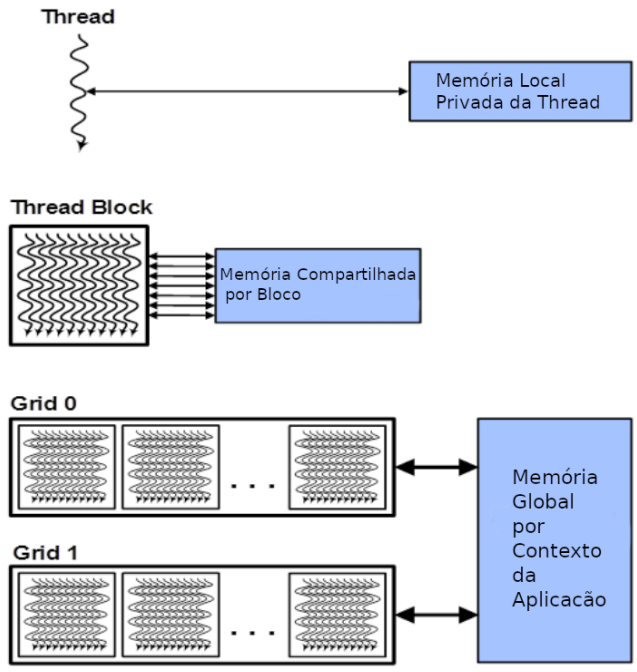

Figura 2.25: Modelo de programação CUDA

Cada *thread* dentro de um *bloco de thread* executa uma instância do *kernel*. Cada *thread* também possui IDs de *thread* e de bloco dentro de seu *bloco de thread* e grade, um contador de programa, registradores, memória privada por *thread*, entradas e resultados de saída.

Um *bloco de thread* é um conjunto de *threads* em execução simultânea cooperando entre si por meio de sincronização de barreira e memória compartilhada. Um *bloco de thread* possui um ID de bloco em sua grade. Uma grade é uma matriz de *blocos de thread* que executam o mesmo *kernel*, leem as entradas da memória global, gravam os resultados para a memória global e fazem a sincronização entre chamadas de *kernel* dependentes.

No modelo de programação paralela CUDA, cada *thread* tem um espaço de memória privado por *thread* usado para salvamento de registradores, chamadas de função e variáveis de arranjo automáticas em C. Cada *bloco de thread* tem um espaço de memória compartilhado por bloco usado para comunicação *inter-*

thread, compartilhamento de dados e compartilhamento de resultados em algoritmos paralelos. Grades de *blocos de thread* compartilham resultados em um espaço de memória global após a sincronização global em todo o *kernel*.

A hierarquia de *threads* do CUDA é mapeada para uma hierarquia de processadores na GPU; uma GPU executa uma ou mais grades de *kernel*; um multiprocessador de fluxo (SM) executa um ou mais *blocos de threads*; e os núcleos CUDA e outras unidades de execução de instruções no SM executam as *threads*. O SM executa *threads* em grupos de 32 *threads* chamados *warps*. Embora os programadores possam ignorar os *warps* para uma execução funcionalmente correta, o desempenho das aplicações pode ser muito melhorado, mantendo-se as *threads* de um mesmo *warp* executando o mesmo código e realizando acessos de memória com endereços próximos.

Uma grande dificuldade encontrada na programação com uso de aceleradores é a movimentação de dados entre a memória do computador hospedeiro e a memória do acelerador, como visto na Figura 2.26.

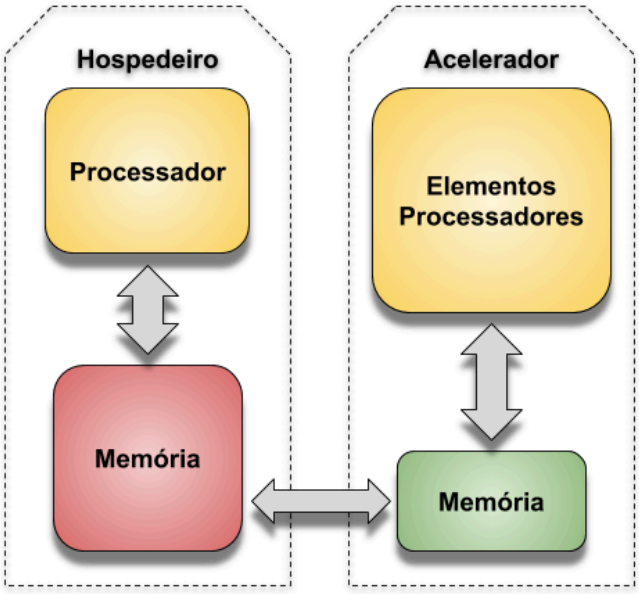

Figura 2.26: Aceleradores

Se não houver uma quantidade significativa de processamento para ser realizada com os dados transferidos, os ganhos obtidos com o processamento podem não compensar o tempo gasto com a transferência dos dados e resultados entre o hospedeiro e o acelerador. Veremos mais detalhes sobre isso na Seção 7.3.

2.6 Exercícios propostos

1. Com relação ao conceito de **processo**, responda:
    1. Quais são as principais informações mantidas dentro de um contexto de processo?
    2. Quais os passos envolvendo a criação de um processo?
    3. Quais são os passos envolvidos na troca de processos?
2. Com relação ao conceito ***thread***, responda:
    1. O que é uma *thread*?
    2. Quais são as principais informações mantidas dentro de um contexto de uma *thread*?
    3. Quais são as principais diferenças entre processos e *threads*?
3. Com relação ao paradigma de programação em memória compartilhada, responda:
    1. Como é feita a comunicação em sistemas com memória compartilhada?
    2. Quando e porquê é utilizada a sincronização por exclusão mútua?

3. Qual o impacto dos mecanismos de sincronização no desempenho dos programas paralelos?

4. Com relação às definições básicas de paralelismo responda:

    1. Qual a distinção entre concorrência e paralelismo?

    2. O que é uma condição de corrida e como o uso da seção ou região crítica pode solucionar esse problema?

    3. Qual a utilidade do uso da sincronização por exclusão mútua?

    4. As condições de corrida podem ocorrer apenas em computadores paralelos?

5. Em relação aos mecanismos reais usados para fornecer sincronização e proteção contra condições de corrida, responda:

    1. Qual a utilidade das operações atômicas?

    2. Qual a definição e quando são utilizados *spin locks*?

    3. Qual a definição e quando são utilizados os semáforos?

6. Uma aplicação dispara duas *threads*, T1 e T2, para execução (códigos mostrados abaixo).

    1. Quais são os possíveis valores esperados na saída do programa (impressos na linha 7)?

    2. Pode ocorrer da *thread* T2 ficar executando indefinidamente, ou seja, não chegar a executar a linha 8?

```
int x=0;                    // Variáveis globais
float item;
void *T1 (void *) {
    for (int i = 1; i < 100; i++) { // Linha 1
        item = ProduzItem();        // Linha 2
```

```
        x++;                            // Linha 3
    }
}
void *T2 (void *) {
int y;
    while(1) {                          // Linha 4
        if ((x%10) == 0) {              // Linha 5
            y=x;                        // Linha 6
            printf("x = %d\n", y);      // Linha 7
            break;                      // Linha 8
        }
    }
}
```

7. Com relação às ocorrências de *deadlock*, responda:

    1. O que é e quais as condições necessárias para a ocorrência de um *deadlock*?

    2. Como a condição de exclusão mútua pode ser removida?

    3. Como a condição de obter e esperar pode ser evitada?

    4. Como a condição de espera circular pode ser evitada?

8. Com relação ao paradigma de programação por troca de mensagens, responda:

    1. Como é feita a sincronização em sistemas de troca de mensagens?

    2. Quais as diferenças entre a comunicação síncrona e assíncrona?

    3. Quais as vantagens do uso da comunicação assíncrona?

    4. Quais os cuidados que devem ser tomados quando do uso de primitivas de comunicação (envio e recepção) não bloqueantes?

5. É possível a ocorrência de *deadlock* em sistemas de troca de mensagem?

6. Qual a finalidade da utilização da sincronização de barreira em sistemas de troca de mensagem?

9. Com relação ao particionamento de dados e tarefas na criação de um programa paralelo, responda:

    1. Quais os cuidados a serem tomados na fase de alocação de tarefas aos processadores?

    2. Quais os cuidados na alocação de tarefas aos processos ou *threads*?

    3. Quais as duas maneiras básicas de particionar o trabalho computacional entre as tarefas?

    4. Qual a relação que deve ser mantida entre as tarefas de comunicação, sincronização e computação para otimizar o tempo de execução dos programas paralelos?

    5. O que deve ser feito se a rede de comunicação utilizada apresentar custos de comunicação variáveis?

10. Com relação aos métodos de balanceamento de carga, responda:

    1. Quais as diferenças entre balanceamento estático e dinâmico?

    2. Em que casos deve ser empregado um ou outro?

    3. Qual a principal indicação de que a carga de trabalho não está adequadamente distribuída entre os processadores?

    4. Qual a aplicação e quais as deficiências do método de balanceamento de saco de tarefas?

5. Quais as características do método de simulação *N-body*?

6. Quais as características do método de refinamento adaptativo de malhas?

11. Com relação à avaliação de desempenho, responda;

    1. Qual a definição de *speed-up* e eficiência?

    2. O que é escalabilidade em um sistema paralelo? Quais são os pontos devem ser considerados em uma análise de escalabilidade?

    3. Qual o *speed-up* máximo teórico segundo a Lei de Amdahl? Quais são os seus pressupostos básicos?

    4. Qual o *speed-up* máximo teórico segundo a Lei de Gustafson? Quais são os seus pressupostos básicos?

    5. Quais são as principais causas da ocorrência de um *speedup* sublinear?

    6. Quais são as principais causas da ocorrência de um *speedup* superlinear?

12. Uma aplicação sequencial pode ser dividida em três etapas: entrada de dados, processamento e saída de dados. A entrada de dados consome 1/8 do tempo total de execução, o processamento 3/4 e a saída de dados 1/8.

    1. Qual será o ganho máximo de desempenho da aplicação se a etapa de processamento for paralelizada e a aplicação for executada em uma máquina com 8 processadores?

    2. Idem para com 800 processadores?

13. Um programa leva 40s para executar em um sistema com um único processador. Responda:

    1. Quantos processadores deverão ser utilizados para reduzir o seu tempo de execução para Math input error, se a eficiência é 0,5?

    2. Qual foi o *speedup* final obtido?

14. Um programa tem uma fração sequencial equivalente a 5% do tempo total de execução. Responda:

    1. Segundo a Lei de Amdahl, qual é o *speedup* máximo que pode ser obtido com a paralelização deste programa?

    2. Qual é o limite de *speedup* para o mesmo programa com o uso de 100 processadores, segundo a Lei de Gustafson?

15. Quais os níveis de paralelismo possíveis na execução de uma aplicação? Detalhe.

16. Em relação às arquiteturas paralelas, responda:

    1. Quais são as características das arquiteturas *multicore* ou CMP?

    2. Como se caracterizam as arquiteturas com aceleradores? Pesquise quais tipos de bibliotecas, ambientes e/ou linguagens podem ser utilizados na sua programação.

    3. Quais são as características das arquiteturas multiprocessadores com memória compartilhada? Pesquise quais tipos de bibliotecas, ambientes e/ou linguagens podem ser utilizados na sua programação.

    4. Quais são as características das arquiteturas multicomputadores com memória distribuída? Pesquise quais tipos de bibliotecas, ambientes e/ou linguagens podem ser utilizados na sua programação.

17. Quais os tipos de fluxos e arquiteturas considerados pela classificação de Flynn?

18. Com relação aos tipos de arquitetura definidos pela classificação de Flynn, responda:

    1. Quais as características dos processadores vetoriais? Pesquise sobre exemplos históricos e atuais desse tipo de arquitetura.

    2. Quais as características dos processadores SIMD convencionais? Pesquise exemplos históricos desse tipo de arquitetura.

    3. Quais as características das arquiteturas sistólicas? Pesquise onde encontramos suas aplicações modernamente.

19. Com relação aos tipos de arquitetura definidos pela classificação de Flynn, responda:

    1. Quais as características das arquiteturas MIMD com memória distribuída? Quais as suas vantagens e desvantagens?

    2. Quais as características das arquiteturas MIMD com memória compartilhada? Quais as suas vantagens e desvantagens?

20. Quais as principais características e limitações das arquiteturas do tipo UMA? Pesquise exemplos modernos desse tipo de arquitetura.

21. Quais as principais características e limitações das arquiteturas do tipo NUMA? Pesquise exemplos modernos desse tipo de arquitetura.

22. Enumere e descreva os principais tipos de arquiteturas de aceleradores.

23. Com relação à arquitetura dos aceleradores gráficos, responda:

    1. Qual a função e quais são os principais componentes de um *streaming multiprocessor*?

    2. O que é uma *warp*?

    3. Como está organizada a arquitetura de memória de um acelerador gráfico?

    4. O que é um bloco de *thread*? Quais as suas principais características?

---

1. Adaptado de (WILKINSON and ALLEN 1998)↩

2. Adaptado de (WILKINSON and ALLEN 1998)↩

3. Obtida em https://www.serdarkorkut.com/wp-content/uploads/2017/06/mesh10.jpg↩

4. Essa lei recebeu o nome depois que Gene Amdahl apresentou-a em um artigo na *AFIPS Spring Joint Computer Conference* em 1967.↩

5. Adaptado de (WILKINSON and ALLEN 1998)↩

3 Comunicação ponto a ponto no MPI

Neste capítulo apresentamos a estrutura principal de um programa em MPI; as funções básicas do MPI; e as rotinas de comunicação ponto a ponto.

3.1 Introdução

Vamos iniciar nosso estudo sobre o desenvolvimento de aplicações paralelas com o MPI mostrando como um programa MPI é construído e quais são as rotinas mais elementares utilizadas em todos os programas.

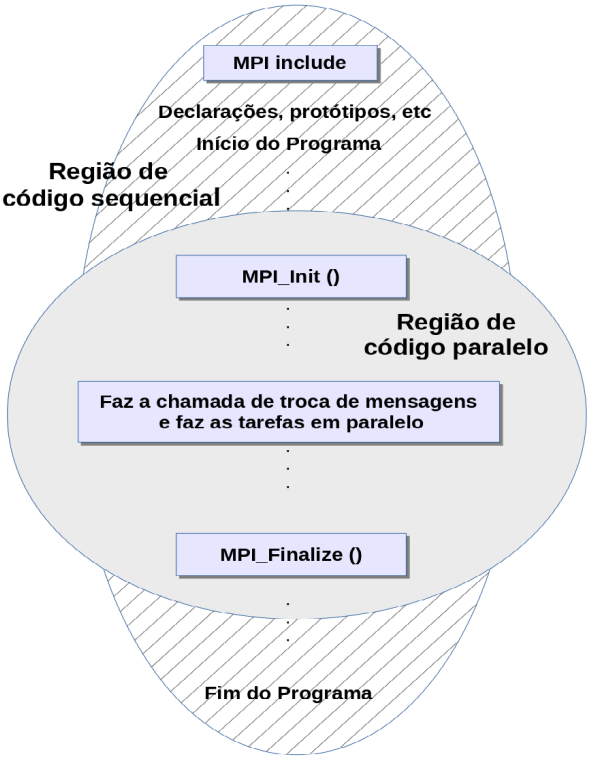

Figura 3.1: Estrutura básica de um programa MPI

A Figura 3.1 mostra a estrutura básica de um programa MPI. O programa pode conter sequências intercaladas de código sequencial e paralelo. A sequência de código paralelo é executada dentro do ambiente MPI, caracterizado por chamadas explícitas às funções de início e término desse ambiente.

O código a seguir apresenta a estrutura ilustrada na Figura 3.1, usando a linguagem C. O arquivo **mpi.h** contém as definições, macros e protótipos de funções necessários para a compilação de um programa MPI.

```
#include "mpi.h"
...
int main(int argc, char *argv[]) {

/* Seção de código sequencial */
```

```
...
/* Nenhuma função MPI pode ser chamada antes deste ponto */
    MPI_Init(&argc, &argv);

/* Seção de código paralelo */
...
    MPI_Finalize();
/* Nenhuma função MPI pode ser chamada depois deste ponto */
/* Seção de código sequencial */
...
    return(0);
}
```

Antes de qualquer função MPI possa ser utilizada, a função **MPI_Init** deve ser chamada. Seus argumentos são os ponteiros para os parâmetros do programa principal, **argc** e **argv**. Essa função permite ao sistema realizar as operações iniciais necessárias para a utilização da biblioteca MPI.

Ao término do trecho de código paralelo, a função **MPI_Finalize** deve ser chamada. Essa função limpa qualquer pendência deixada pelo MPI, por exemplo, de recepções pendentes ainda não completadas.

Mas antes de nos aprofundarmos no estudo dessas rotinas, vamos primeiro estudar o conceito de comunicadores utilizado pelo MPI.

3.2 Comunicadores

A biblioteca MPI usa o conceito de **comunicador**, ilustrado na Figura 3.2, para definir o universo de processos envolvidos em uma operação de comunicação, por meio de atributos de **grupo** e **contexto**:

- Grupo: conjunto ordenado de processos contidos no comunicador, cuja ordem é chamada de ranque (*rank*).
- Contexto: um rótulo (número inteiro único) definido pelo sistema, diferenciando um comunicador do outro.

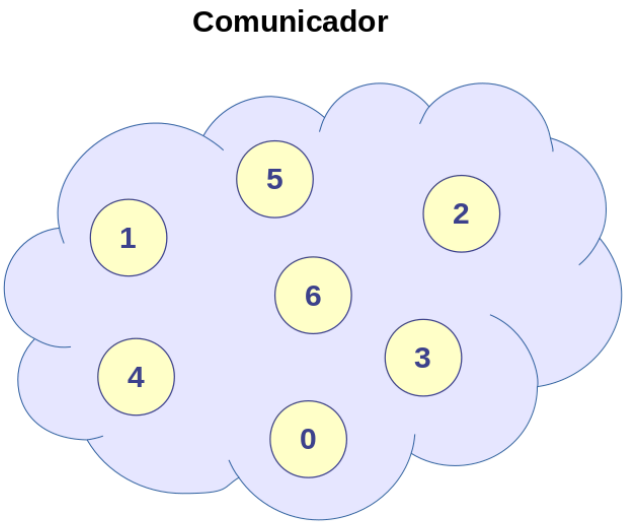

Figura 3.2: Comunicador

Todos os processos de um mesmo grupo e que tenham o mesmo contexto pertencem a um mesmo comunicador e podem se comunicar diretamente. O comunicador padrão utilizado pelo MPI é o **MPI_COMM_WORLD**, que consiste no conjunto de todos os processos criados no início da execução de um programa.

Cada processo em um comunicador é identificado unicamente pelo seu **ranque**, um número positivo entre 0 e Math input error, onde Math input error é o número total de processos em execução no comunicador. A função **MPI_Comm_Rank** serve para determinar o **ranque** de um processo, retornando o número de ordem do processo dentro do comunicador. Vejamos os detalhes da sua sintaxe:

int MPI_Comm_rank(MPI_Comm com, int *meu_ranque)

O primeiro argumento *com* é um comunicador. Para os programas básicos, o único comunicador necessário é **MPI_COMM_WORLD**. O segundo argumento, *meu_ranque*, retorna o valor do **ranque** do processo. Como dito, dentro de um grupo com Math input error processos, os ranques recebem números inteiros positivos entre 0 e Math input error. Se um processo é associado a mais de um comunicador, ele terá um ranque único em cada comunicador.

Muitas construções nos programas MPI também dependem do número de processos executando concorrentemente uma tarefa. O MPI oferece a função **MPI_Comm_size** para determinar esse valor. Essa função retorna no seu segundo argumento o número de processos em um dado comunicador, como o **MPI_COMM_WORLD**. A sua sintaxe é:

int MPI_Comm_size(MPI_Comm com, int *num_procs)

Vamos apresentar em seguida um primeiro exemplo de programa MPI, que faz uso de rotinas básicas de controle de execução e informação utilizadas pelo MPI.

3.3 Exemplo de um programa em MPI

Apresentamos a seguir a listagem de um programa bem simples em MPI. Quando esse programa é executado são criados *num_procs* processos.

O processo com ranque igual a 0 recebe Math input error mensagens dos demais processos, contendo seus respectivos ranques, e imprime cada uma das mensagens recebidas.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_simples.c  */
int meu_ranque, num_procs;
int origem, destino = 0, etiq = 1;
char mensagem[200];
MPI_Status estado;
    /* Chamada inicial para o MPI */
    MPI_Init(&argc, &argv);
    /* Determina o ranque e número de processos em execução */
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Todos processos com ranque diferente de 0 enviam uma mensagem */
    if (meu_ranque != destino) {
```

```
        sprintf(mensagem, "Processo %d está vivo!", meu_ranque);
        MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, destino, etiq, MPI_COMM_WORLD);
    } else {
    /* Processo com ranque 0 recebe num_procs-1 mensagens */
        for (origem = 1; origem < num_procs; origem++) {
             MPI_Recv(mensagem, 200, MPI_CHAR, origem, etiq, MPI_COMM_WORLD, &estado);
    /* Imprime as mensagens recebidas */
             printf("%s\n",mensagem);
        }
    }
    MPI_Finalize();
    return(0);
}
```

Para compilar esse programa, podemos salvá-lo em um arquivo com o nome, por exemplo, *mpi_simples.c* e usar em seguida o seguinte comando em um terminal de linha de comando em um sistema Posix:

```
$ mpicc -o simples mpi_simples.c
```

Para executar o programa, use o seguinte comando:

```
$ mpirun -np 4 ./simples
```

O resultado da execução poderá variar de acordo com a ordem de execução dos processos. Uma possível saída é mostrada a seguir:

```
Processo 1 está vivo!
Processo 2 está vivo!
Processo 3 está vivo!
```

Nas seções seguintes, as funções utilizadas nesse programa são apresentadas em detalhes.

3.4 Funções de gerenciamento do ambiente

Esse grupo de rotinas é usado para estabelecer e obter informações sobre o ambiente de execução do MPI e cobre uma variedade de propósitos, tais como, iniciar e terminar o ambiente MPI, verificar qual o ranque do processo e a versão da biblioteca MPI, entre outros.

3.4.1 Iniciando e finalizando o MPI

A função **MPI_Init** inicia o ambiente de execução do MPI. Essa função deve ser chamada em todo programa MPI, antes de qualquer outra função MPI, e apenas uma única vez. Para os programas em C, **MPI_Init** serve para passar os argumentos recebidos na linha de comando pelo programa para todos os processos, embora isso não seja requerido pelo padrão e seja dependente de implementação. Sua sintaxe é:

int MPI_Init (int *argc, char **argv[])

A função **MPI_Finalize** termina a execução do ambiente MPI e por isso deve ser a última função MPI a ser chamada em todo programa MPI. Nenhuma outra rotina MPI pode ser chamada depois dela. Sua sintaxe é:

int MPI_Finalize (void)

Em algumas situações, pode ser necessário verificar se as funções **MPI_Init** e **MPI_Finalize** já foram chamadas. Para isso existem as funções **MPI_Initialized** e **MPI_Finalized**. O trecho de código seguinte ilustra o uso dessas funções.

```
#include "mpi.h"
int main(int argc, char *argv[]) {
int iniciado, finalizado;
...
    MPI_Initialized(&iniciado);
    if (!iniciado)
        MPI_Init(&argc, &argv);
    /* Realiza o trabalho em paralelo */
...
    /* Quando o programa está para terminar */
    MPI_Finalized(&finalizado);
    if (!finalizado)
        MPI_Finalize();
    return(0);
}
```

A rotina **MPI_Initialized** indica se a função **MPI_Init** foi chamada, retornando um valor lógico verdadeiro (1) ou falso (0). Como o MPI requer que a função **MPI_Init** seja chamada apenas uma vez por cada processo, isso pode gerar um problema para módulos ou bibliotecas que precisam fazer uso do MPI e chamar a rotina **MPI_Init**, se necessário. A função **MPI_Initialized** resolve esse problema, sendo a única rotina que pode ser chamada antes de **MPI_Init**. Sua sintaxe é a seguinte:

int MPI_Initialized (int *flag)

A rotina **MPI_Finalized** indica se a função **MPI_Finalize** foi chamada. A sintaxe dessa função é:

int MPI_Finalized (int *flag)

3.4.2 Medindo o tempo de execução

A função **MPI_Wtime** retorna o tempo total em segundos, em precisão dupla, decorrido desde um instante determinado no passado. Esse instante é dependente de implementação, mas deve ser sempre o mesmo para uma dada implementação. A sua sintaxe é definida como:

double MPI_Wtime (void)

A função **MPI_Wtick** retorna a resolução em segundos (em precisão dupla) da função **MPI_Wtime**. Por exemplo, se o relógio for implementado pelo *hardware* como um contador incrementado a cada milissegundo, então o valor retornado por **MPI_Wtick** deverá ser Math input error. Sua sintaxe é definida a seguir:

double MPI_Wtick (void)

3.4.3 Abortando um programa

A função **MPI_Abort** termina todos os processos associados ao comunicador. Na maioria das implementações, essa função termina todos os processos independentemente do comunicador que foi especificado. A sua sintaxe é a seguinte:

int MPI_Abort(MPI_Comm com, int erro)

3.4.4 Verificando a versão da biblioteca MPI

A função **MPI_Get_version** retorna a versão e a subversão do padrão MPI implementado pela biblioteca utilizada. Vejamos a sua sintaxe:

int MPI_Get_version(int *versao, int *subversao)

3.4.5 Verificando o nome do processador

A função **MPI_Get_processor_name** retorna o nome do processador e o tamanho (comprimento) do seu nome. O vetor de caracteres apontado por *nome* deve ter capacidade para, no mínimo, o número de caracteres especificado em **MPI_MAX_PROCESSOR_NAME**.

O valor exato retornado em *nome* é dependente da implementação — pode não ser exatamente igual à saída dos comandos "hostname" ou "host" do Posix. A sintaxe dessa função é:

int MPI_Get_processor_name (char *nome, int *comprimento)

O código do próximo exemplo mostra o uso dessas novas funções de gerenciamento do ambiente MPI.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_funcoes.c  */
int meu_ranque, num_procs;
int versao, subversao,  aux, ret;
double tempo_inicial, tempo_final;
char maquina[MPI_MAX_PROCESSOR_NAME];
    /* Inicia o MPI. Em caso de erro aborta o programa */
    ret = MPI_Init(&argc, &argv);
    if (ret != MPI_SUCCESS) {
        printf("Erro ao iniciar o programa MPI. Abortando.\n");
        MPI_Abort(MPI_COMM_WORLD, ret);
    }
    tempo_inicial = MPI_Wtime();
    /* Imprime a versão e subversão da biblioteca MPI */
    MPI_Get_version(&versao,&subversao);
    printf("Versão do MPI  = %d Subversão = %d \n", versao, subversao);
    /* Obtém o ranque e número de processos em execução */
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    /* Recebe o nome do computador onde o processo é executado */
    MPI_Get_processor_name(maquina, &aux);
      printf("Número de tarefas = %d Meu ranque = %d Executando em  %s\n", num_procs,
meu_ranque, maquina);
    tempo_final = MPI_Wtime();
      printf("Foram gastos %3.6f segundos com precisão de %3.3e segundos\n",tempo_final-
tempo_inicial, MPI_Wtick());
    /* Finaliza o MPI */
    MPI_Finalize();
    return(0);
}
```

A execução desse programa com 4 processos poderá ter a seguinte saída como resultado:

```
Versão do MPI  = 3 Subversão = 1
Número de tarefas = 4 Meu ranque = 2 Executando em  maquina002
```

```
Foram gastos 0.000028 segundos com precisão de 1.000e-09 segundos
Versão do MPI  = 3 Subversão = 1
Número de tarefas = 4 Meu ranque = 1 Executando em  maquina001
Foram gastos 0.000024 segundos com precisão de 1.000e-09 segundos
Versão do MPI  = 3 Subversão = 1
Número de tarefas = 4 Meu ranque = 0 Executando em  maquina000
Foram gastos 0.000024 segundos com precisão de 1.000e-09 segundos
Versão do MPI  = 3 Subversão = 1
Número de tarefas = 4 Meu ranque = 3 Executando em  maquina003
Foram gastos 0.000020 segundos com precisão de 1.000e-09 segundos
```

3.5 Envio e recepção de mensagens

As operações básicas de troca de mensagens nos programas MPI são executadas pelas funções **MPI_Send** e **MPI_Recv**. A primeira envia a mensagem para um determinado processo e a segunda é utilizada para receber uma mensagem.

As duas funções são bloqueantes, isto é, a rotina de envio só retorna depois que todos os dados forem copiados das variáveis utilizadas para o envio da mensagem e a rotina de recepção só retorna quando os dados recebidos estiverem disponíveis nas variáveis utilizadas nas rotinas de recepção.

O uso ou não de *buffers* pela biblioteca do MPI depende do tamanho da mensagem enviada. Ambas são funções bloqueantes, mas podem ser síncronas ou não, dependendo de um tamanho máximo de mensagem, tamanho esse que é dependente da implementação. Maiores detalhes a respeito dos modos de envio e recepção de mensagens do MPI podem ser vistos na Seção 5.5.

Uma mensagem no MPI é uma estrutura constituída pelo **envelope** e os **dados** a serem enviados de um processo para outro. O **envelope** de uma mensagem possui algumas informações adicionais para a transmissão da mensagem com sucesso. No MPI, o envelope de uma mensagem contém os seguintes campos, como ilustrado na Figura 3.3.

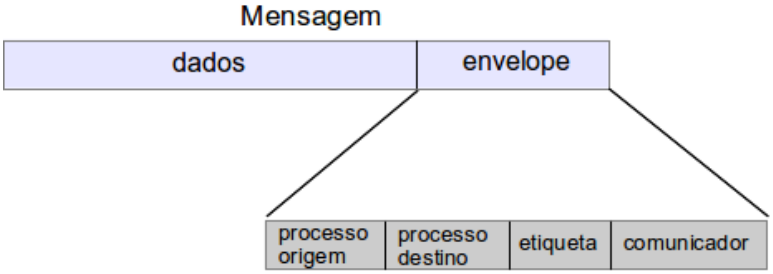

Figura 3.3: Conteúdo de uma mensagem no MPI

- Envelope da mensagem:
  - o ranque do processo origem ;
  - o ranque do processo destino;
  - uma etiqueta (*tag*) especificando o tipo da mensagem;
  - um comunicador (domínio de comunicação).

O padrão MPI define que as mensagens enviadas por um determinado processo não devem ultrapassar umas às outras. Por exemplo, suponha que o **processo 0** envie duas mensagens,

primeiro Math input error e depois Math input error. Se o **processo 2** chamar duas rotinas de recepção que podem receber qualquer uma das duas mensagens, a ordem de entrega das mensagens será preservada; ou seja, Math input error será entregue antes de Math input error independentemente da ordem de chegada das duas mensagens na máquina destino, conforme podemos observar na Figura 3.4.

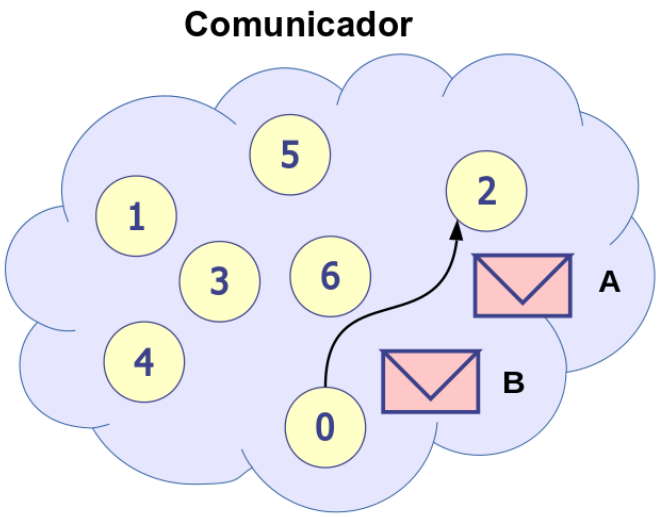

Figura 3.4: Ordem de entrega das mensagens

Em outras palavras, se um processo invocar duas operações de recepção em sequência, ambas podendo receber quaisquer das mensagens enviadas, então a primeira operação de recepção postada vai receber a primeira mensagem enviada pelo processo emissor.

Um detalhe importante: a regra de ordenação das mensagens não se aplica se houver mais de uma *thread* participando das operações de comunicação dentro de um mesmo processo, uma vez que a ordem de execução das *threads* vai determinar a ordem das chamadas das funções de envio e recepção.

Outro detalhe em relação à ordem de entrega das mensagens: se dois processos distintos enviam uma mensagem do mesmo tipo para o mesmo processo destino e esse processo invoca a operação de recepção apenas uma vez, o MPI não garante qual das mensagens será recebida.

Por exemplo, vamos supor que o **processo 0** envia uma mensagem para o **processo 2** e, por circunstância, o **processo 1** também envia uma mensagem competidora para o mesmo **processo 2**. Se o **processo 2** estiver esperando apenas uma mensagem, isso implica que apenas uma das operações de envio se completará. A outra operação ficará pendente e, dependendo do tipo de operação de envio, o processo emissor poderá ficar bloqueado indefinidamente.

3.5.1 Enviando uma mensagem

A função **MPI_Send** copia os dados a serem enviados de uma variável (um vetor normalmente) situada na memória do processo origem. A localização, o tamanho e o tipo de dados contidos nesse vetor são especificados pelos três primeiros parâmetros da função: *mensagem*, *cont* e *tipo_mpi*. Adicionalmente, a operação de envio especifica um envelope para a mensagem, com o destino da mensagem e algumas informações adicionais, a etiqueta e o comunicador, utilizadas pela operação

de recepção para selecionar uma mensagem em particular. Então, os três últimos parâmetros da função (*destino*, *etiq* e *com*), mais o ranque do processo de origem, especificam o envelope para a mensagem enviada.

int MPI_Send(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino, int etiq, MPI_Comm com)

O protótipo da função tem os seguintes parâmetros:

- mensagem: endereço inicial dos dados a serem enviados.
- cont: quantidade de dados a serem enviados.
- tipo_mpi: o tipo MPI dos dados a serem enviados:
  MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- destino: *ranque* do processo destino.
- etiq: etiqueta da mensagem.
- com: comunicador utilizado.

O argumento *com* deve ser preenchido com o nome do comunicador correspondente aos processos emissor e receptor. O comunicador padrão **MPI_COMM_WORLD** será utilizado por enquanto.

3.5.2 Recebendo uma mensagem

A mensagem a ser recebida é selecionada de acordo com o valor do seu envelope, sendo armazenada em uma variável (usualmente um vetor) de recepção na aplicação. Os três primeiros parâmetros (*mensagem*, *cont* e *tipo_mpi*) da função de recepção especificam a localização, o tamanho e o tipo de dados do vetor de recepção. Os próximos três parâmetros (*origem*, *etiq* e *com*) são usados para selecionar a mensagem que será recebida. O último parâmetro *estado* é uma estrutura utilizada para retornar informações específicas sobre a mensagem recebida, sendo útil quando são utilizados "coringas (*)" para a sua recepção. Maiores detalhes podem ser obtidos na Seção 3.6.

int MPI_Recv(void* mensagem, int cont, MPI_Datatype tipo_mpi, int origem, int etiq, MPI_Comm com, MPI_Status* estado)

O protótipo da função tem os seguintes parâmetros:

- mensagem: endereço inicial do vetor de recepção
- cont: número máximo de dados a serem recebidos
- tipo_mpi: o tipo MPI dos dados a serem recebidos:
  MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- origem: ranque do processo origem (* = **MPI_ANY_SOURCE**)
- etiq: etiqueta da mensagem (* = **MPI_ANY_TAG**)

- com: comunicador utilizado
- estado: estrutura com três elementos: **MPI_SOURCE, MPI_TAG, MPI_ERROR**.

A mensagem recebida será aquela enviada pelo processo com ranque e etiqueta idênticos aos definidos nos argumentos *origem* e *etiq*. O tamanho do vetor de recepção apontado pelo argumento *mensagem* deve ser maior ou igual ao tamanho da mensagem recebida. Caso contrário, a função retornará com erro. O MPI permite que uma mensagem seja recebida enquanto houver espaço suficiente de memória alocado.

O argumento *origem* é o ranque do processo que enviou a mensagem. O MPI permite que esse parâmetro seja preenchido com um curinga, indicando que qualquer processo poderá ser o emissor da mensagem esperada. O MPI define a constante **MPI_ANY_SOURCE** para ser usada como curinga para o parâmetro *origem*, sendo que não há coringa para o argumento *destino*.

O argumento *etiq* deve ser um valor inteiro utilizado para identificar o tipo da mensagem esperada, sendo que esse parâmetro também pode ser preenchido com um coringa. Nesse caso, mensagens com qualquer valor de etiqueta poderão ser recebidas. O MPI define a constante **MPI_ANY_TAG** para ser usada como coringa para o parâmetro *etiq*.

Assim como na função **MPI_Send**, o argumento *com* deve ser preenchido com o nome do comunicador correspondente aos processos emissor e receptor. Não existe curinga para o comunicador.

3.5.3 Correspondência entre os tipos MPI e C

Na Tabela 3.1 observamos uma lista de tipos do MPI e sua correspondência com os tipos em C. Os dois últimos tipos, **MPI_BYTE e MPI_PACKED** não possuem correspondência com os tipos padrões em C. O uso do tipo **MPI_BYTE** dispensa a conversão quando os tipos de dados são diferentes no envio e na recepção. O tipo **MPI_PACKED** é utilizado para dados empacotados com a rotina **MPI_Packed**, mas que não será abordada neste livro.

Tabela 3.1: Correspondência entre os tipos de dados em C e MPI

Tipo de dados em MPI	Tipo de dados em C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t

Tipo de dados em MPI	Tipo de dados em C
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

3.6 Identificando as mensagens recebidas

As operações de comunicação ponto a ponto do MPI tipicamente envolvem a troca de mensagens entre dois processos diferentes do MPI. Um dos processos realiza a operação de envio e o outro a operação correspondente de recepção.

Alguns parâmetros podem ser usados pelo processo receptor para distinguir entre as diversas mensagens recebidas. O argumento *origem* serve para identificar as mensagens recebidas de diferentes processos. Já o parâmetro *etiq* é um inteiro não negativo especificado pelo usuário para distinguir entre as mensagens provenientes de um mesmo processo.

O MPI garante que inteiros entre 0 e 32767 podem ser usados como etiquetas, mas algumas implementações permitem valores maiores. O maior valor admissível para etiqueta é disponível através do atributo **MPI_TAG_UB**.

Assim, para que o **processo 1** possa enviar uma mensagem para o **processo 2**, os argumentos que o **processo 1** usa em **MPI_Send** devem ser idênticos aos que o **processo 2** usa em **MPI_Recv**.

3.6.1 Utilizando as informações em MPI_Status

Uma informação mais detalhada sobre as mensagens recebidas com o uso de coringas é retornada pela função **MPI_Recv** no manipulador (*handle*) *MPI_Status*. O último argumento (*estado*) dessa função faz referência a uma estrutura com três elementos: um para a origem, o segundo para a etiqueta e um terceiro para a informação de erro, como ilustrado na Tabela 3.2.

Então, por exemplo, se o valor preenchido para o parâmetro *origem* na recepção foi **MPI_ANY_SOURCE**, o elemento *MPI_SOURCE* indica o *ranque* do processo emissor da mensagem. Da mesma forma, se o valor do parâmetro *etiq* da recepção foi preenchido com **MPI_ANY_TAG**, o elemento *MPI_TAG* contém a *etiqueta* usada para o envio da mensagem. O elemento *MPI_ERROR* dessa estrutura contém o código de erro da mensagem recebida.

Há dois ponteiros do tipo *MPI_Status* predefinidos utilizados para ignorar a informação recebida: **MPI_STATUS_IGNORE** e **MPI_STATUSES_IGNORE**, esse último aplicável em funções que retornam um vetor de *MPI_Status*.

Tabela 3.2: Informações em MPI_Status

Informação	**MPI_Status**
Remetente	estado.MPI_SOURCE
Etiqueta	estado.MPI_TAG
Código de Erro	estado.MPI_ERROR

Além das informações sobre emissor e etiqueta da mensagem extraídas dos elementos da estrutura *estado*, é possível extrair também a quantidade de itens recebidos na mensagem por meio da rotina **MPI_Get_count**, cuja sintaxe é:

int MPI_Get_count(MPI_Status *estado, MPI_Datatype tipo_mpi, int *total)

O número total de elementos recebidos na mensagem será retornado na variável *total*. O programa a seguir serve para ilustrar o uso das informações de *estado*.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define MAX 100
int main(int argc, char *argv[]) { /* mpi_status.c */
int meu_ranque, total_num, etiq = 0;
int origem=0, destino=1, numeros[MAX];
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    if (meu_ranque == origem) {
    /* Envia um número aleatório de inteiros para o processo 1 */
        srand(MPI_Wtime());
        total_num = (rand() / (float)RAND_MAX) * MAX;
    /* Envia a quantidade de inteiros para o processo 1 */
        MPI_Send(numeros, total_num, MPI_INT, destino, etiq, MPI_COMM_WORLD);
        printf("Processo %d enviou %d números para 1\n", origem, total_num);
    }
    if (meu_ranque == destino) {
    /* Recebe no máximo MAX números do processo 0 */
        MPI_Recv(numeros, MAX, MPI_INT, origem, etiq, MPI_COMM_WORLD, &estado);
    /* Determina quantos números foram realmente recebidos */
        MPI_Get_count(&estado, MPI_INT, &total_num);
    /* Imprime a informação no manipulador "estado" */
         printf("Processo %d recebeu %d números. Origem da mensagem = %d, etiqueta = %d\n",
destino, total_num, estado.MPI_SOURCE, estado.MPI_TAG);
    }
    MPI_Finalize();
    return(0);
}
```

A seguir vemos um possível resultado da execução desse programa:

```
Processo 0 enviou 58 números para 1
Processo 1 recebeu 58 números. Origem da mensagem = 0, etiqueta = 0
```

Como vemos, o **processo 0** envia até MAX números inteiros aleatoriamente para o **processo 1**. O **processo 1** chama **MPI_Recv** para receber um total de MAX números inteiros. Embora o **processo 1** esteja passando MAX números como argumento para **MPI_Recv**, ele vai receber apenas a quantidade de números enviados de fato pelo **processo 0**. No código, o **processo 1** chama **MPI_Get_count**, usando **MPI_INT** como tipo de dado, para determinar quantos inteiros foram realmente recebidos. Junto da impressão do tamanho da mensagem recebida, o **processo 1** também imprime a origem e a etiqueta da mensagem acessando os elementos **MPI_SOURCE** e **MPI_TAG** na estrutura *estado*, do tipo **MPI_Status**.

É importante observar que o valor retornado por **MPI_Get_count** é relativo ao tipo de dados MPI passado como segundo argumento da função. Se tivéssemos usado **MPI_CHAR** como tipo de dado, a quantidade retornada deveria ser quatro vezes maior (assumindo-se que um tipo inteiro tem quatro *bytes* e um tipo *char* apenas um *byte*).

3.6.2 Verificando o tamanho das mensagens recebidas

Agora que já vimos como o objeto *MPI_Status* funciona, podemos utilizá-lo junto com a rotina **MPI_Probe** para determinar o tamanho de uma mensagem antes de efetivamente recebê-la. Isso nos permite dimensionar a variável de recepção adequadamente, em vez de reservar um espaço exageradamente grande, para todos os possíveis tamanhos de mensagem. O protótipo da função é o seguinte:

int MPI_Probe(int origem, int etiq, MPI_Comm com, MPI_Status* estado)

A função **MPI_Probe** é muito parecida com a função **MPI_Recv**. Em realidade, a primeira realiza as mesmas funções que a segunda, menos receber a mensagem. A função **MPI_Probe** bloqueará esperando por uma mensagem com a origem e etiqueta correspondentes. Quando a mensagem estiver disponível, ela vai preencher a estrutura *estado* com a informação apropriada. O usuário pode então utilizar a função **MPI_Recv** para receber a mensagem verdadeira. O uso desta função é detalhado a seguir.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int main(int argc, char *argv[]) { /* mpi_probe.c  */
int meu_ranque, num_procs, etiq=0;
int* vet_num;
int total_num, origem=0, destino=1, numeros[MAX];
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Envia um número aleatório de dados */
    if (meu_ranque == origem) {
        srand(MPI_Wtime());
        total_num = (rand() / (float)RAND_MAX) * MAX;
        MPI_Send(numeros, total_num, MPI_INT, destino, etiq, MPI_COMM_WORLD);
          printf("Processo %d enviou %d números  para processo %d.\n", origem, total_num,
destino);
    }
    else
        if (meu_ranque == destino) {
    /* Verifica se chegou mensagem do processo origem */
            MPI_Probe(origem, etiq, MPI_COMM_WORLD, &estado);
    /* Obtém o tamanho da mensagem com o manipulador "estado" */
```

```
            MPI_Get_count(&estado, MPI_INT, &total_num);
    /* Aloca um vetor para guardar os números transmitidos */
            vet_num = (int*) malloc(sizeof(int)*total_num);
    /* Agora recebe a mensagem com o vetor previamente alocado  */
                    MPI_Recv(vet_num, total_num, MPI_INT, origem, etiq, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
         printf("Processo %d recebeu dinamicamente %d números de %d.\n", destino, total_num,
origem);
            free(vet_num);
          }
    MPI_Finalize();
    return(0);
}
```

Nesse exemplo, de forma similar ao exemplo com uso das informações de estado, o **processo 0** escolhe um número aleatório de números para enviar para o **processo 1**. Porém, nesse novo exemplo o **processo 1** chama **MPI_Probe** para descobrir quantos elementos o processador 0 está tentando enviar, com ajuda da função **MPI_Count**. O **processo 1** então aloca um vetor com o tamanho apropriado e recebe os números. O resultado da execução do programa seria similar a este:

```
Processo 0 enviou 93 números  para processo 1.
Processo 1 recebeu dinamicamente 93 números de 0.
```

Embora esse exemplo seja trivial, a função **MPI_Probe** tem grande utilidade em aplicações do tipo mestre/trabalhador onde há grande troca de mensagens de tamanho variável.

3.7 Algumas recomendações

A seguir, listamos alguns erros comuns cometidos por programadores menos experientes e elencamos algumas recomendações para serem utilizadas quando programando com as rotinas de envio e recepção ponto a ponto do MPI.

Erros frequentes

- Tentar receber dados antes que uma mensagem de envio correspondente tenha sido postada;
- Utilizar parâmetros incorretos ou inconsistentes nas rotinas de envio e recepção de mensagens;
- Esperar que os argumentos *argv* e *argc* sejam passados automaticamente para todos os processos;
- Utilizar rotinas MPI antes do **MPI_Init()** ou depois do **MPI_Finalize()**;
- Note que a sequência exata das computações e comunicações pode variar de execução para execução e de um processador para outro. Portanto, deve-se ter cuidado com relação ao que será assumido na ordem de execução das operações e mensagens no programa.

Otimizando o desempenho

- Tente sempre minimizar a comunicação entre os processos. Há três aspectos principais em relação a isso:

- Tente sempre limitar a quantidade de dados transferidos entre processos. A largura de banda disponível entre os processos é um recurso limitado.

- Tente reduzir o número de transações envolvidas na transferência de dados, já que a latência na transmissão de mensagens no MPI pode ser significativa. Podemos agrupar várias mensagens em uma só, já que isso tem menor sobrecarga. Tipicamente, em um *cluster*, pacotes com menos de 1 KB são ineficientes, sendo que pacotes maiores que 10 KB são transmitidos de forma mais eficiente.

- Se um processo precisa de dados que estão com outro processo, isso pode requerer que esse processo fique em espera, diminuindo o ganho total da execução em paralelo da aplicação. Assim, pode ser interessante organizar o seu programa de forma que todos os processos se comuniquem ao mesmo tempo.

- Embora a rotina **MPI_Probe** seja bastante conveniente, evite seu uso sempre que o desempenho seja um requisito. Seu uso implica que o sistema deve alocar recursos suficientes para receber a mensagem e depois copiá-la para os *buffers* de usuário. Isso é muito lento. Talvez seja mais eficiente enviar duas mensagens: a primeira contendo o tamanho da segunda mensagem, que efetivamente terá a mensagem real.

3.8 Estudo de caso: método do trapézio

Nesta seção, encontramos um exemplo prático de um programa com as rotinas de comunicação ponto a ponto do MPI. O nosso estudo de caso será o cálculo do valor de uma integral definida utilizando o método do trapézio.

A Figura 3.5 ilustra o método do trapézio, que estima o valor de f(x) dividindo o intervalo Math input error em Math input error segmentos iguais e calculando a seguinte equação:

Math input error

Onde:

Math input error Math input error e Math input error

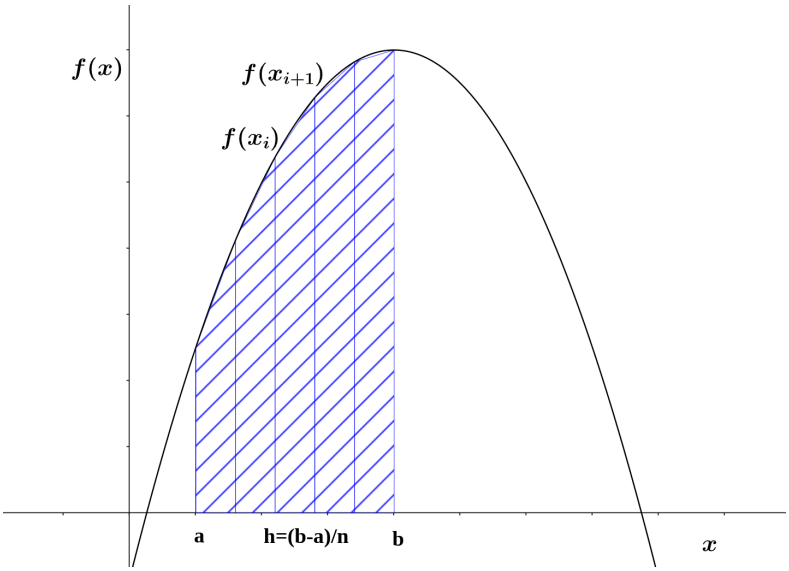

Figura 3.5: Cálculo da integral usando o método do trapézio

Considerando Math input error como uma função já predefinida, o código apresentado a seguir é uma possível versão sequencial para o cálculo da integral definida, utilizando o método dos trapézios.[1]

```
#include <stdio.h>
#include <math.h>
float f(float x) {    /* Calcula f(x). */
float return_val;
   return_val=exp(x);
   return return_val;
}
int main(int argc, char *argv[]) {  /* trapezio_seq.c  */
float integral;   /* integral armazena resultado final */
float a, b;       /* a, b -  limite esquerdo e direito da função */
int   i,n;        /* n - número de trapezóides */
float x, h;       /* h - largura da base do trapezóide */

   printf("Entre a, b, e n \n");
   scanf("%f %f %d", &a, &b, &n);
   h = (b-a)/n;
   integral = (f(a) + f(b))/2.0;
   x = a;
   for (i = 1; i < n; i++) {
        x += h;
        integral += f(x);
   }
   integral *= h;
   printf("Com n = %d trapezóides, a estimativa \n", n);
   printf("da integral de %f até %f = %f\n", a, b, integral);
   return(0);
}
```

Esse programa pode ser paralelizado facilmente com a decomposição ou particionamento estático dos dados, assim, cada um dos Math input error processos ficará responsável por calcular o valor da integral em um subconjunto consecutivo de Math input error trapezoides. Para determinar o valor total da integral, os valores parciais calculados localmente por cada processo deverão ser

enviados para um único processo, o qual efetuará a soma desses valores parciais e imprimirá o resultado final. Nesse caso, o custo de comunicação entre os processos será relativamente baixo, e os valores *speedup* e eficiência obtidos serão muito altos. Um número de trapezoides (Math input error) distribuídos para cada processo suficientemente grande é o único requisito para permitir uma carga de trabalho adequada em cada processador.

Essa forma de divisão, contudo, necessita que Math input error seja um múltiplo inteiro de Math input error. Se essa exigência não for respeitada, haverá uma dificuldade para a distribuição de um número igual de trapézios para todos os processos. O último processo ficará com o resto dessa divisão e processará um número maior de trapézios. Como resultado, o tempo de execução em paralelo não será ótimo, e isso se agrava com o aumento do número de trapézios e processos em execução.

Uma forma alternativa de particionamento dos dados é determinar que o primeiro processo calcule a área do primeiro trapézio, o segundo processo calcule a área do segundo trapézio e assim por diante. Após calcular a área do primeiro trapézio, o primeiro processo vai calcular a área de todos os trapézios distantes Math input error trapézios entre si no intervalo [a, b]. Essa nova forma de divisão dos dados tem a vantagem de que Math input error não precisa ser múltiplo de Math input error e a distribuição de carga para cada processo é mais equânime. Dessa forma, cada processo Math input error calculará aproximadamente um valor definido pela equação:

Math input error

Onde:

Math input error, Math input error, Math input error

Cada processo calculará o somatório do valor da função Math input error, sendo Math input error, Math input error e Math input error, onde Math input error é o ranque do processo. Esse cálculo será feito em até, mas não exatamente, Math input error pontos por cada processo e o total desse somatório será multiplicado por Math input error, cabendo ainda ao processo com ranque igual a 0 calcular:

Math input error

Logo, cada processo precisa das seguintes informações:

- Seu ranque, Math input error;
- O número de processos, Math input error;
- O intervalo inteiro de integração, Math input error;
- O número de subintervalos, Math input error.

Os dois primeiros itens, Math input error e Math input error, podem ser encontrados chamando as funções MPI:

```
MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

Os dois últimos itens devem ser informados pelo usuário. Para a nossa primeira tentativa de paralelização, vamos definir valores fixos para esses itens, atribuídos no código do programa. No exemplo utilizado, vamos calcular o valor aproximado da integral da função Math input error no intervalo entre 0 e 1, com Math input error. Esse valor de Math input error é necessário para garantir um tempo mínimo de execução para cada processo e vai exigir precisão dupla nas variáveis que calculam o valor da integral, para não haver perda de precisão no resultado, cujo valor aproximado é 1,71828182846.

Após calcular a soma de todos os valores locais, os processos enviarão seus resultados para o processo 0 e esse processo fará a soma final. Como já dito, a forma de paralelização utilizada não requer que Math input error seja necessariamente múltiplo de Math input error, como podemos ver no exemplo apresentado a seguir.

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
double f(double x) {
        double return_val;
        return_val = exp(x); /* Função exponencial */
        return return_val;
}
int main(int argc, char *argv[]) { /* mpi_trapezio.c  */
int meu_ranque, num_procs;  /* respectivamente q e p */
double a = 0.0,  b = 1.0;   /* Limites da integral */
double tempo_inicial, tempo_final; /* Tempo de execução */
long int n = 100000000;     /* Número de trapezoides */
double x, h;                /* x e h, a base do trapezoide */
double integral=0.0, total; /* Integral de cada processo e total */
int origem, destino = 0;    /* Origem e destino das mensagens */
int etiq = 3;               /* Uma etiqueta qualquer */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* h é o mesmo para todos os processos */
    h = (b - a)/n;
    /* O processo 0 calcula o valor de f(x)/2 em a e b */
    if (meu_ranque == 0) {
        tempo_inicial = MPI_Wtime();
        integral = (f(a) + f(b))/2.0;
    }
    /* Cada processo calcula a integral sobre n/num_procs trapézios */
    for (x = a+h*(meu_ranque+1); x < b ; x += num_procs*h) {
         integral += f(x);
    }
    integral = integral*h;
    /* O processo 0 soma as integrais parciais recebidas */
    if (meu_ranque == 0) {
        total = integral;
        for (origem = 1; origem < num_procs; origem++) {
                      MPI_Recv(&integral,  1,  MPI_DOUBLE,  origem,  etiq,  MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
             total += integral;
        }
    } else {
     /* As integrais parciais são enviadas para o processo 0 */
        MPI_Send(&integral, 1, MPI_DOUBLE, destino, etiq, MPI_COMM_WORLD);
    }
    /* Imprime o resultado */
    if (meu_ranque == 0) {
```

```
        tempo_final = MPI_Wtime();
        printf("Foram gastos %3.1f segundos\n",tempo_final-tempo_inicial);
        printf("Com n = %ld trapezoides, a estimativa\n", n);
        printf("da integral de %lf até %lf = %lf \n", a, b, total);
    }
    MPI_Finalize();
    return(0);
}
```

Esse programa deve ser salvo com o nome *mpi_trapezio.c*. Em seguida, deve ser compilado (não esquecer a opção *-lm*) e executado com os seguintes comandos em um terminal de um sistema operacional do tipo Posix:

```
$ mpicc -o trapezio mpi_trapezio.c -lm -O3
$ mpirun -np 4 ./trapezio
```

O resultado da execução desse programa será o seguinte:

```
Foram gastos 0.6 segundos
Com n = 100000000 trapezoides, a estimativa
da integral de 0.000000 até 1.000000 = 1.718282
```

O código possui um alto potencial de paralelismo, tendo em vista que os únicos valores transmitidos para o processo com ranque igual a 0 são os valores calculados localmente em cada processo (variável *integral*). O código foi instrumentado com a rotina **MPI_Wtime()** (Seção 3.4) para medir o tempo total de execução do programa e neste exemplo é recomendável que a chamada dessa rotina seja feita apenas pelo processo com ranque 0, o último a terminar. A partir dos valores obtidos com a execução em sistemas com diversos processadores físicos, é possível obter a curva de *speedup* para esse programa.

3.8.1 Avaliação de desempenho

A avaliação de desempenho deste programa foi dividida em duas abordagens: a primeira delas analisou o desempenho sob a ótica de Amdahl e a segunda sob a de Gustafson, sendo que nos dois casos o número de *processos* (e processadores) variou entre 1 e 24. Na primeira análise variamos apenas o número de *processos*, mas mantendo-se fixo o tamanho do problema. Na segunda abordagem, o tamanho do problema foi aumentado à medida que o número de *processos* crescia, esperando-se com isso obter um tempo constante de execução para solução do problema.

Um computador com uma arquitetura de multiprocessamento simétrico e um total de 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz) foram utilizados para este estudo de escalabilidade.

A Tabela 3.3 apresenta os tempos de execução, métricas de *speedup* e eficiência que foram obtidos com Math input error igual a 128.000.000.000 e usando-se como referência a execução com um processo e tempo de execução de 2353 segundos. Aumentamos o valor de Math input error pois o tempo de computação total para o valor original foi muito pequeno, dificultando uma correta medição dos tempos e cálculo das métricas de *speedup* e eficiência.

Tabela 3.3: Métricas do método do trapézio - Amdahl

Processos	**Tempo Total**	***Speedup***	**Eficiência**
1	2353 s	1,0	1,00
2	1215 s	1,9	0,97

Processos	Tempo Total	*Speedup*	Eficiência
4	608 s	3,9	0,97
8	334 s	7,0	0,88
12	233 s	10,1	0,84
16	173 s	13,6	0,85
20	141 s	16,6	0,83
24	119 s	19,8	0,82

A Figura 3.6 mostra o gráfico com a evolução do *speedup* conforme aumenta-se a quantidade de processos, de acordo com os dados obtidos a partir da Tabela 3.3. A linha contínua nesse gráfico indica o *speedup* linear ideal e a outra linha com pontos destacados apresenta o resultado do *speedup* calculado.

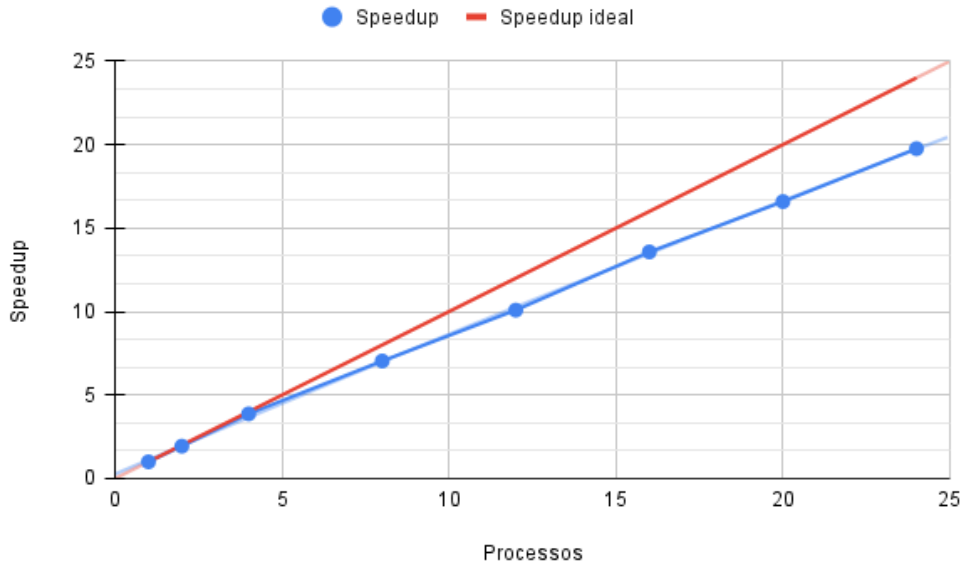

Figura 3.6: Gráfico de *speedup* - método do trapézio - Amdahl

Já a eficiência pode ser observada no gráfico da Figura 3.7. que começa com um valor perto do ótimo, mas decai gradativamente até se estabilizar em torno de 0,80 ou 80%.

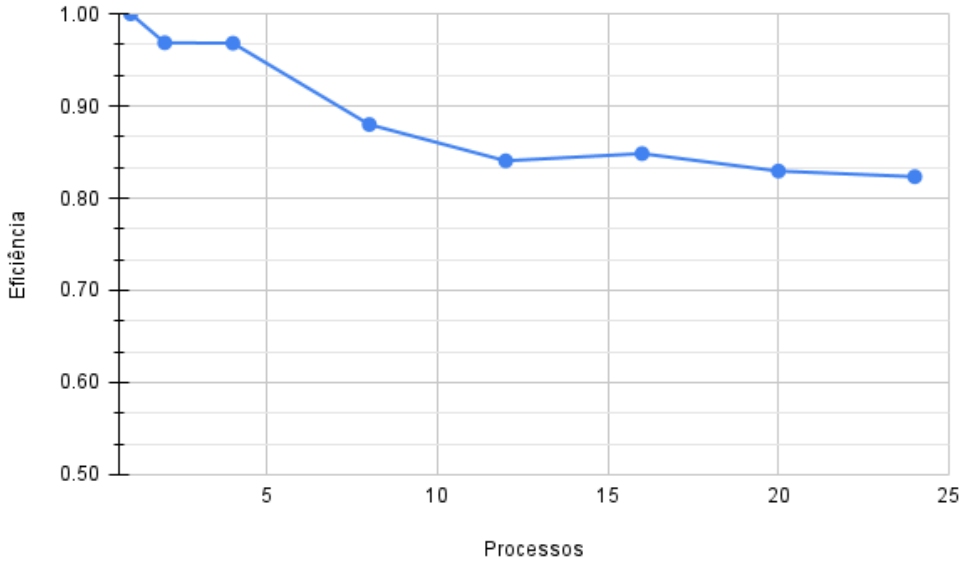

Figura 3.7: Gráfico de eficiência - método do trapézio - Amdahl

Essa queda de desempenho ocorre principalmente pela espera que ocorre no processo com ranque 0 pelos resultados parciais enviados pelos demais processos. Particularmente, como a divisão de Math input error por Math input error não é exata em muitos casos, isso deixa alguns processos com algumas iterações a mais do laço para serem calculadas. Os processos têm então tempos de execução distintos, já que a distribuição de carga não é perfeita, e o processo mais lento determina o tempo total de execução do programa.

Se avaliarmos o desempenho de acordo com a definição de Gustafson (veja a Seção 2.4.3), devemos aumentar o tamanho do problema proporcionalmente ao aumento do número de processos.

O aumento do tamanho do problema, no caso do trapézio, está diretamente relacionado à quantidade de iterações do laço de repetição. Portanto, ao aumentarmos linearmente o valor de Math input error, aumentamos proporcionalmente a carga de trabalho.

Nesse experimento, definimos inicialmente Math input error para Math input error processo. Seguimos aumentando o tamanho do problema em razão da quantidade de processos utilizados, conforme mostra a Tabela 3.4.

Tabela 3.4: Métricas método do trapézio - Gustafson

Tamanho N	**Tempo Seq.**	**Processos (*p*)**	**Tempo Paralelo**	**Speedup**	**Efici- ência**
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Tamanho N	Tempo Seq.	Processos (*p*)	Tempo Paralelo	Speedup	Efici- ência
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Esses resultados são sumarizados no gráfico da Figura 3.8, onde a linha contínua indica o *speedup* linear ideal e a linha logo abaixo com pontos indica o resultado do *speedup* calculado.

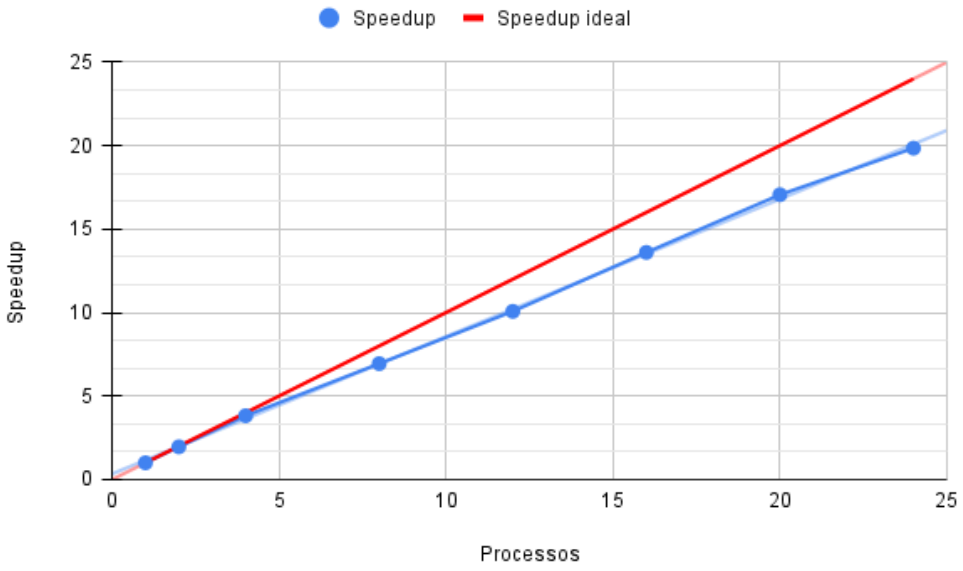

Figura 3.8: Gráfico de *speedup* - método do trapézio - Gustafson

Os resultados obtidos em ambos métodos de avaliação, Amdahl e Gustafson, são muito similares, tanto para os gráficos de *speedup*, quanto para os gráficos de eficiência (veja também a Figura 3.9).

Isso era de alguma forma esperado, já que para os tamanhos de problema que foram avaliados, o maior impacto na perda de desempenho se deve ao desbalanceamento de carga, e não à fração de processamento sequencial do programa, que é muito pequena.

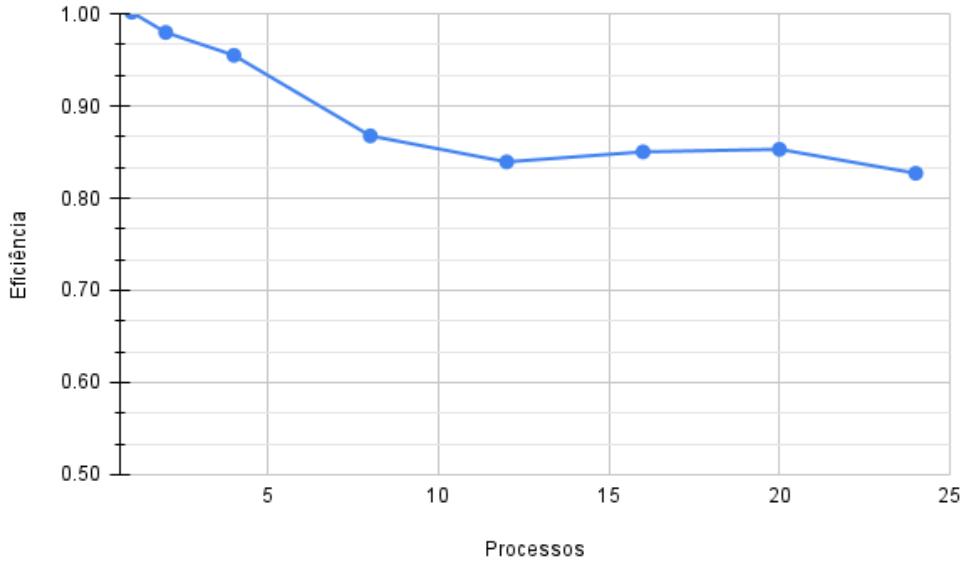

Figura 3.9: Gráfico de eficiência - método do trapézio - Gustafson

3.9 Exercícios propostos

1. Quais as diferenças entre as seguintes rotinas:
    1. **MPI_Comm_rank** e **MPI_Comm_ size**?
    2. **MPI_Init** e **MPI_Initialized**?
    3. **MPI_Finalize** e **MPI_Finalized**?
2. A função **MPI_Abort** vai finalizar todos os processos do comunicador ou apenas aquele que a chamou? O que acontece com os processos de outros comunicadores?
3. Como se definir adequadamente o tamanho da cadeia de caracteres para ser utilizada com a rotina **MPI_Get_processor_name**?
4. O que se entende do fato de as rotinas de envio e recepção padrão do MPI (**MPI_Send** e **MPI_Recv**) serem bloqueantes?
5. Com relação às mensagens no MPI, responda:
    1. Qual o conteúdo do envelope de uma mensagem MPI?
    2. O que se pode e o que não é possível garantir em relação à ordem de entrega das mensagens no MPI?
    3. Em um programa MPI correto, qual a relação entre o tamanho da mensagem de recepção em uma função **MPI_Recv** e o tamanho da mensagem enviada?
6. Com relação à rotina de recepção **MPI_Recv**, responda:

1. Quais são os "coringas" utilizados para indicar o processo de origem e a etiqueta?

2. Quais são os seus componentes e para que serve a estrutura do tipo MPI_Status?

3. Como para fazer para ignorar a informação que é devolvida na estrutura MPI_Status?

7. Considere o seguinte trecho de código:

```
MPI_Comm_size(MPI_WORLD_COMM, num_procs);
do {
    MPI_Send(mensagem,strlen(mensagem)+1,MPI_CHAR,
             num_procs-1,1,MPI_COMM_WORLD);
   } while (--num_procs);
```

8. Escreva quatro versões diferentes de trechos de código usando a rotina **MPI_Recv** para receber essa mensagem com sucesso. Justifique a sua resposta para cada caso.

9. Com relação ao primeiro exemplo da Seção 3.3 (mpi_simples.c), responda:

    1. O que acontece caso a chamada da função **MPI_Comm_rank** seja movida para antes da chamada da função **MPI_Init**?

    2. Caso o tamanho do vetor de recepção *mensagem* seja menor que o tamanho da mensagem enviada, o que acontece?

    3. Modifique esse exemplo para que no laço do processo 0 sejam recebidas mensagens de qualquer processo, com qualquer etiqueta. O que será impresso na saída do processo 0?

10. Escreva um trecho de código para identificar o processo que enviou a mensagem e imprimir o número total de elementos recebido pela rotina a seguir.

```
MPI_Recv(vetor, MAX_FLOAT, MPI_FLOAT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &estado);
```

11. Modifique o código da Seção 3.6, com a função **MPI_Probe**, para execução com um número qualquer de processos, e que todos os processos com ranque diferente de 0 enviem uma mensagem com tamanho aleatório para o processo 0, que irá receber apenas uma mensagem, mas usando um coringa como fonte e etiqueta (MPI_ANY_SOURCE e MPI_ANY_TAG). Pode-se determinar exatamente qual saída será impressa pelo processo 0? A execução do programa termina normalmente?

12. Execute o programa paralelo do trapézio com 1, 2, 4, 6 e 8 processos em uma máquina com 8 processadores físicos (núcleos). Trace os gráficos com as curvas de tempo de execução, *speedup* e eficiência em função do número de processos.

13. Escreva um programa para calcular o produto escalar de dois vetores. Utilize rotinas **MPI_Send** e **MPI_Recv** para comunicação entre os processos. Considere cada vetor com Math input error posições e divida a operação entre Math input error processos distintos. Considere que a divisão de Math input error por Math input error não tem resto. Atribua os valores iniciais aos vetores localmente e assuma que todos processos conhecem o tamanho Math input error total do vetor.

14. Execute o programa do exercício anterior com N = 100.000.000 e com 1, 2, 4, 6 e 8 processos em uma máquina com 8 processadores. Instrumente o código com uso da rotina

MPI_Wtime. Trace os gráficos com as curvas de tempo de execução, *speedup* e eficiência em função do número de processos.

15. Considere um anel com *num_procs* processos. Escreva um programa utilizando rotinas **MPI_Send e MPI_Recv** para comunicação entre os processos que faça circular uma mensagem contendo um inteiro positivo ao longo desse canal. O processo com ranque igual a 0 é quem inicia a transmissão e cada vez que a mensagem passa por ele novamente o valor contido na mensagem deve ser decrementado de um até chegar ao valor 0. Quando um processo receber a mensagem com valor 0 ele deverá passá-la adiante e então terminar a sua execução.

    Dica: para identificar os nós vizinhos no anel faça:
    dest = (ranque+1)%num_procs;
    fonte = (ranque+num_procs-1)%num_procs.

---

1. Adaptado de (Pacheco 1996)↩

4 Comunicação coletiva no MPI

Neste capítulo apresentamos as rotinas de comunicação coletiva, diversos exemplos para seu uso, estudos de caso e recomendações de boas práticas de programação utilizando rotinas de comunicação coletivas.

Um padrão de comunicação envolvendo todos os processos em um comunicador é chamado de comunicação coletiva. As funções de comunicação coletiva visam oferecer as seguintes operações:

- Sincronização com **barreira** (Seção 4.1);
- Comunicação global:
  - **Difusão** (Seção 4.2);
  - **Distribuição** (Seção 4.3);
  - **Coleta** (Seções 4.4, 4.7 e 4.8);
- **Redução** global (Seções 4.5 e 4.6).

A característica fundamental dessas operações é possuírem um "grupo de processos" envolvidos. Entretanto, em vez de definir identificadores de grupos explícitos, o MPI usa o **comunicador** como argumento em todas as operações. O comunicador funciona ao mesmo tempo como identificador do grupo e como domínio de comunicação. Os mesmos comunicadores usados com as funções de comunicação ponto a ponto podem ser usados para as funções de comunicação coletiva.

As operações de comunicação coletiva são mais restritivas que as comunicações ponto a ponto, possuindo as características enumeradas a seguir:

- a quantidade de dados enviados deve corresponder exatamente à quantidade de dados especificada pelo receptor;
- apenas a versão bloqueante das funções está disponível[1];
- o argumento *etiq* não existe;
- as funções estão disponíveis apenas no modo "padrão".

O MPI assegura que as mensagens geradas para as comunicações coletivas não serão confundidas com as mensagens geradas para a comunicação ponto a ponto (Marc et al. 1996). Veremos que, de forma diferente das funções de comunicação ponto a ponto, todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis. Quando uma operação coletiva possui um único processo de origem ou um único processo de destino, esse processo é chamado de **raiz**.

Detalhes sobre operações de envio e recepção bloqueantes foram apresentados na Seção 2.2.2 e maiores informações sobre o modo padrão de comunicação do MPI estão disponíveis na seção seguinte.

4.1 Barreira

A função de barreira bloqueia o processo, que fica esperando por todos os processos do mesmo grupo. A sua sintaxe é muito simples:

int MPI_Barrier(MPI_Comm com)

A função **MPI_Barrier** fornece um mecanismo para sincronizar todos os processos no comunicador *com*. Cada processo bloqueia (i.e., suspende a sua execução), esperando todos os processos

associados ao mesmo comunicador chamarem a função **MPI_Barrier**. A Figura 4.1 ilustra o funcionamento do mecanismo de barreira.

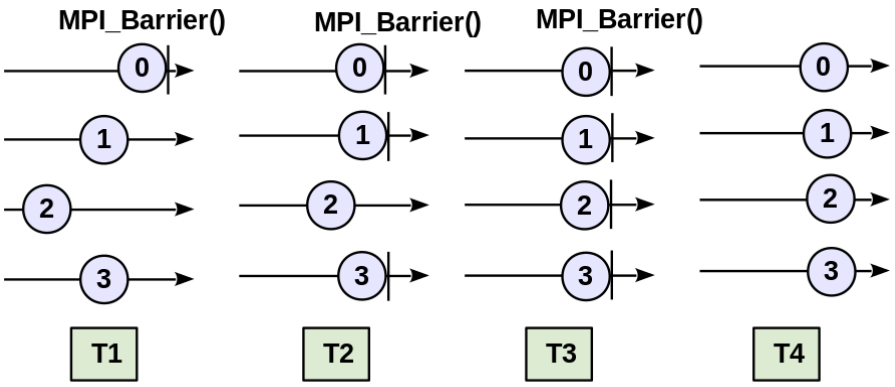

Figura 4.1: Operação de barreira

O exemplo a seguir apresenta o uso da função **MPI_Barrier**. Neste exemplo todos os processos só podem prosseguir e passar da barreira, após o processo com ranque igual a 0 fazer a leitura da console e alcançar a barreira.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) { /* mpi_barrier.c  */
int meu_ranque, num_procs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD,&meu_ranque);

    if (meu_ranque == 0 ) {
        printf ("Estou atrasado para a barreira! \n");
        getchar();
    }
    MPI_Barrier(MPI_COMM_WORLD);
     printf("Passei da barreira. Eu sou o %d de %d processos
\n", meu_ranque, num_procs);
    MPI_Finalize();
    return(0);
}
```

Um possível resultado para a execução desse programa com quatro processos seria a saída a seguir.

```
Estou atrasado para a barreira!

Passei da barreira. Eu sou o 1 de 4 processos
Passei da barreira. Eu sou o 2 de 4 processos
Passei da barreira. Eu sou o 3 de 4 processos
Passei da barreira. Eu sou o 0 de 4 processos
```

Uma causa frequente de erros de programação no uso da função de barreira é a ocorrência de *deadlock* (veja a Seção 2.2.1) quando algum dos processos do comunicador não chama a função corretamente. Se um ou mais processos associados ao comunicador não chamarem a função **MPI_Barrier** o mesmo número de vezes que os demais processos do comunicador, a operação de barreira não será concluída e os processos que chegarem à barreira ficarão esperando indefinidamente pelos demais.

Vemos a seguir alguns exemplos de uso correto e incorreto da função **MPI_Barrier**:

```
/* 1 - Trecho correto  */
    if (meu_ranque != 0) {
        ...
        MPI_Send(...);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    else {
        for (origem = 1; origem < num_procs; origem++) {
             MPI_Recv(...);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
/*  2- Uso incorreto  */
    if (meu_ranque == 0) {
        MPI_Barrier(MPI_COMM_WORLD);
    }
    else {
         printf("Este caminho não contém uma barreira\n");
    }
/* 3 - Trecho correto */
    if (x < 3) {
```

```
        ...
        MPI_Barrier(MPI_COMM_WORLD);
    }
/* 4- Uso incorreto */
    while(x < meu_ranque) {
          MPI_Barrier(MPI_COMM_WORLD);
          x ++;
    }
```

Há três desvios e um laço apresentados no exemplo:

1. O primeiro desvio (a sentença *if (meu_ranque != 0)*) é livre de erros de sincronização, já que no outro ramo do desvio a função também é chamada.

2. O segundo desvio (*if (meu_ranque == 0)*) contém um erro porque apenas o processo com ranque igual a 0 vai chamar a barreira. Alguns processos executarão a barreira e outros não, provocando a ocorrência de *deadlock*.

3. O desvio seguinte está correto, já que a decisão a ser tomada (*if (x<3)*) não depende do valor do ranque, então podemos esperar que todos os processos se comportem de maneira similar.

4. O laço (*while(x < meu_ranque)*) contém um erro também, já que diferentes processos podem executar um número diferente de iterações e então chamar a função de barreira um número diferente de vezes. Os processos com um número maior de chamadas ficarão presos na barreira.

Maiores detalhes sobre como evitar a ocorrência de *deadlock* com o uso das funções **MPI_Send** e **MPI_Recv** podem ser vistos na Seção 5.6.

4.2 Difusão

Uma difusão (*broadcast*) é uma comunicação coletiva na qual um processo, denominado processo **raiz**, envia o mesmo conjunto de valores para todos os processos do mesmo grupo, inclusive a ele mesmo. No MPI, a função **MPI_Bcast** faz a difusão de uma mesma mensagem do processo **raiz** para todos os processos associados ao comunicador especificado. A sua sintaxe é:

int MPI_Bcast (void *mensagem, int cont, MPI_Datatype tipo_mpi, int raiz, MPI_Comm com)

Essa função simplesmente envia os dados apontados por *mensagem* no processo *raiz*, para todos os processos no comunicador *com*, conforme ilustrado na Figura 4.2. Para que os demais processos do grupo recebam a mensagem, cada um deles deve chamar a mesma função (**MPI_Bcast**) com os mesmos argumentos para os parâmetros *raiz* e *com*.

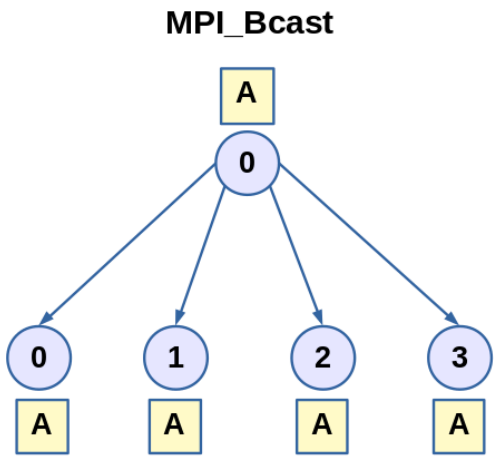

Figura 4.2: Operação de difusão

Uma mensagem de difusão não pode ser recebida com a função **MPI_Recv**. Os parâmetros *cont* e *tipo_mpi*, assim como nas funções **MPI_Send** e **MPI_Recv**, especificam o tamanho da mensagem e o tipo de dado transmitido. Contudo, ao contrário das funções ponto a ponto, o padrão MPI exige que, para uma comunicação coletiva, *cont* e *tipo_mpi* tenham os mesmos valores para todos os processos no mesmo comunicador.

O programa a seguir apresenta o uso da rotina de difusão. O processo com ranque 0 lê um valor da entrada padrão e então chama a função **MPI_Bcast** para enviá-lo para todos os processos do mesmo grupo. Os demais processos chamam a mesma função (**MPI_Bcast**) para receber esse valor. Quando essa função retornar em cada processo, a variável *valor* terá sido alterada com o conteúdo enviado pelo processo com ranque 0.

O que distingue o comportamento da função em cada processo, isto é, qual processo está enviando ou recebendo os dados, é o parâmetro *raiz*. Se esse parâmetro for igual ao ranque do processo, ele está enviando os dados; caso contrário, recebendo.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_bcast.c  */
int valor, meu_ranque, raiz = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    /* Cada processo tem um valor inicial diferente */
    valor = meu_ranque;
    /* O valor a ser enviado é lido pelo processo raiz */
    if (meu_ranque == raiz) {
        printf("Entre um valor: \n");
        scanf("%d", &valor);
    }
    /* A rotina de difusão é chamada por todos processos */
    MPI_Bcast(&valor, 1, MPI_INT, raiz, MPI_COMM_WORLD);
    /* O valor agora é o mesmo em todos os processos */
        printf("O processo com ranque %d recebeu o valor:
%d\n",meu_ranque, valor);
    /* Termina a execução */
    MPI_Finalize();
    return 0;
}
```

Um possível resultado da execução desse programa com 4 processos é mostrado a seguir:

```
Entre um valor:
256
O processo com ranque 0 recebeu o valor: 256
O processo com ranque 1 recebeu o valor: 256
O processo com ranque 2 recebeu o valor: 256
O processo com ranque 3 recebeu o valor: 256
```

O envio de mensagens para todos os processos deve ser feito preferencialmente com o uso da função **MPI_Bcast** porque sua implementação normalmente é bem mais eficiente do que o uso combinado das funções de **MPI_Send** e **MPI_Recv**(Marc et al. 1996).

Uma observação importante com relação à semântica da função **MPI_Bcast**: essa operação pode retornar para o processo que a invocou, assim que sua participação na comunicação coletiva terminar. Isto é, o processo estará liberado para acessar e fazer alterações nas variáveis usadas para a comunicação. Entretanto, não há garantia de que os demais processos também já tenham terminado as suas respectivas operações. Em resumo, a função **MPI_Bcast não** serve para sincronizar os processos de um grupo, sendo que o mesmo vale para as demais funções de comunicação coletiva apresentadas nesta seção.

4.3 Distribuição

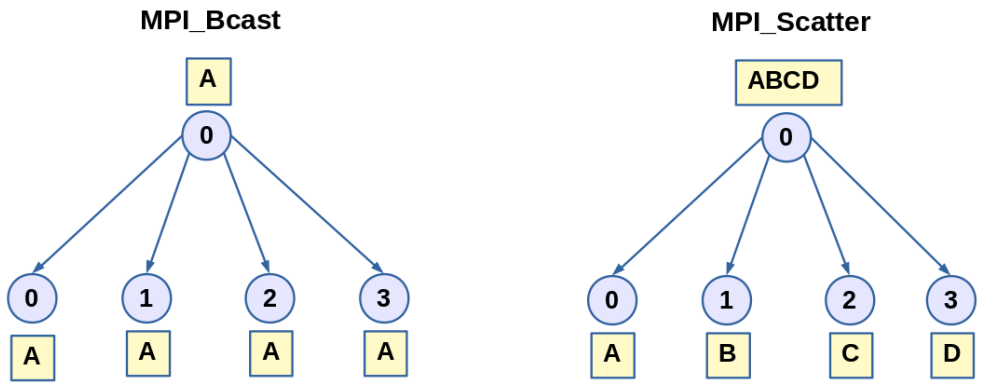

Figura 4.3: Operação de difusão versus distribuição

A operação coletiva de distribuição (*scatter*), ilustrada na Figura 4.3, permite enviar partes distintas de um vetor de dados do processo **raiz** para todos os processos do grupo, inclusive o processo **raiz**. A sintaxe da função **MPI_Scatter** é a seguinte:

int MPI_Scatter(void* vet_envia, int cont_envia, MPI_Datatype tipo_envia, void* vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, int raiz, MPI_Comm com)

O processo com o ranque igual a *raiz* distribui o conteúdo de *vet_envia* entre todos os processos do comunicador, inclusive ele mesmo. O conteúdo de *vet_envia* é dividido em Math input error segmentos, cada um consistindo de *cont_envia* itens. O primeiro segmento vai para o **processo 0**, o segundo para o **processo 1**, e assim sucessivamente na ordem dos ranques. Os argumentos *vet_envia* e *cont_envia* são significativos apenas no processo *raiz*, e o argumento *cont_envia* não é o tamanho do vetor *vet_envia*, mas sim o número de itens enviados do processo *raiz* para cada um dos processos.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define TAM_VET 10
int main(int argc, char *argv[]) { /* mpi_scatter.c  */
int i, meu_ranque, num_procs, raiz = 0;
int *vet_envia, vet_recebe[TAM_VET];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
     /* O processo raiz aloca o espaço de memória e inicia o
vetor */
    if (meu_ranque == raiz) {
                                vet_envia    =    (int*)    malloc
(num_procs*TAM_VET*sizeof(int));
        for (i = 0; i < (TAM_VET*num_procs); i++)
             vet_envia[i] = i;
    }
```

```
      /* O vetor é distribuído em partes iguais entre todos
processos */
        MPI_Scatter(vet_envia, TAM_VET, MPI_INT, vet_recebe,
TAM_VET, MPI_INT, raiz, MPI_COMM_WORLD);
    /* Cada processo imprime a parte que recebeu */
    printf("Processo = %d, recebeu", meu_ranque);
    for (i = 0; i < TAM_VET; i ++)
         printf(" %d", vet_recebe[i]);
    printf("\n");
    /* Termina a execução */
    MPI_Finalize();
    return(0);
}
```

No exemplo apresentado, o processo *raiz* envia partes um vetor com Math input error inteiros para Math input error processos e para ele mesmo, de modo que o próprio processo *raiz* receberá parte do vetor. Vemos a seguir um possível resultado para execução desse programa com 4 processos:

```
Processo =  0, recebeu 0 1 2 3 4 5 6 7 8 9
Processo =  2, recebeu 20 21 22 23 24 25 26 27 28 29
Processo =  3, recebeu 30 31 32 33 34 35 36 37 38 39
Processo =  1, recebeu 10 11 12 13 14 15 16 17 18 19
```

4.4 Coleta

As operações de coleta permitem reunir dados de todos os processos do comunicador realizando a operação inversa da função de distribuição (*scatter*). Na definição básica da operação de coleta (*gather*) cada processo do comunicador, incluindo o próprio processo **raiz**, envia uma mensagem para o processo **raiz**. O processo **raiz** armazena as mensagens recebidas na mesma ordem do ranque dos processos emissores. A Figura 4.4 ilustra esse comportamento.

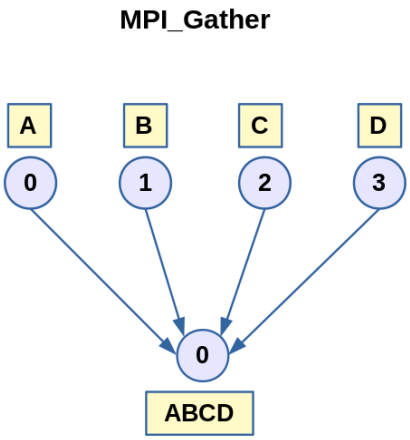

Figura 4.4: Operação de coleta

Vemos em seguida a sintaxe dessa função:

int MPI_Gather(void* vet_envia, int cont_envia, MPI_Datatype tipo_envia, void* vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, int raiz, MPI_comm com)

Cada processo no comunicador *com* envia o conteúdo de *vet_envia*, com *cont_envia* itens, para o processo com *ranque* igual a *raiz*. O processo *raiz* concatena os dados recebidos no vetor *vet_recebe* na ordem definida pelo *ranque* de cada processo. Os argumentos com sufixo *recebe* são significativos apenas no processo com *ranque* igual a *raiz*. Um detalhe importante: o argumento *cont_recebe* indica o número de itens recebidos de **cada** processo, e não o número total de itens recebido pelo processo *raiz*. Normalmente, *cont_envia* é igual a *cont_recebe* e *tipo_envia* é igual a *tipo_recebe*. O uso da função **MPI_Gather** é exemplificado no código a seguir.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define TAM_VET 10
int main(int argc, char *argv[]) { /* mpi_gather.c  */
int meu_ranque, num_procs, i, raiz = 0;
int *vet_recebe,  vet_envia[TAM_VET];
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
     /* O processo raiz aloca memória para o vetor de recepção
*/
    if (meu_ranque == raiz)
                                    vet_recebe    =    (int    *)
malloc(num_procs*TAM_VET*sizeof(int));
    /* Cada processo atribui valor inicial ao vetor de envio */
    for (i = 0; i < TAM_VET; i++)
         vet_envia[i] = meu_ranque;
    /* As partes são recebidas pelo processo raiz em vet_recebe
*/
        MPI_Gather  (vet_envia,  TAM_VET,  MPI_INT,  vet_recebe,
TAM_VET, MPI_INT, raiz, MPI_COMM_WORLD);
    /* O processo raiz imprime o vetor recebido */
    if (meu_ranque == raiz) {
        printf("Processo =  %d, recebeu", meu_ranque);
        for (i = 0; i < (TAM_VET*num_procs); i ++) {
             printf(" %d", vet_recebe[i]);
        }
        printf("\n");
    }
    /* Termina a execução */
    MPI_Finalize();
    return(0);
}
```

No exemplo apresentado, o processo *raiz* monta um vetor com Math input error inteiros recebidos de cada um dos outros Math input error processos e dele mesmo. Note que o próprio processo *raiz* contribui para a montagem do vetor final, cujo tamanho é igual a Math input error, onde Math input error é o número de processos participantes da computação.

O efeito da operação **MPI_Gather** é como se cada um dos Math input error processos de um grupo (incluindo o processo *raiz*) tivesse realizado uma chamada à função **MPI_Send** e o processo *raiz* tivesse realizado Math input error chamadas à função **MPI_Recv**, recebendo as mensagens na ordem dos ranques dos processos. Entretanto, assim como ocorre com a operação de

difusão **MPI_Bcast**, é normalmente mais eficiente usar a função **MPI_Gather** em vez de uma combinação das operações de comunicação ponto a ponto. Um possível resultado da execução desse exemplo com 4 processos seria:

```
Processo =  0, recebeu  0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
```

4.5 Redução

No programa do cálculo da integral pelo método do trapézio, cada processador executa os mesmos comandos até o final da fase de soma. Mas após essa fase as tarefas não são bem balanceadas, ficando a cargo do processo com ranque 0 receber e totalizar todos os valores. Para diminuir esse desbalanceamento, poderíamos utilizar o seguinte procedimento:

1. 1 envia resultado para 0, 3 para 2, 5 para 4, 7 para 6.
2. 0 soma sua integral com a de 1, 2 soma com a de 3, etc.
3. 2 envia para 0, 6 envia para 4.
4. 0 soma, 4 soma.
5. 4 envia para 0.
6. 0 soma

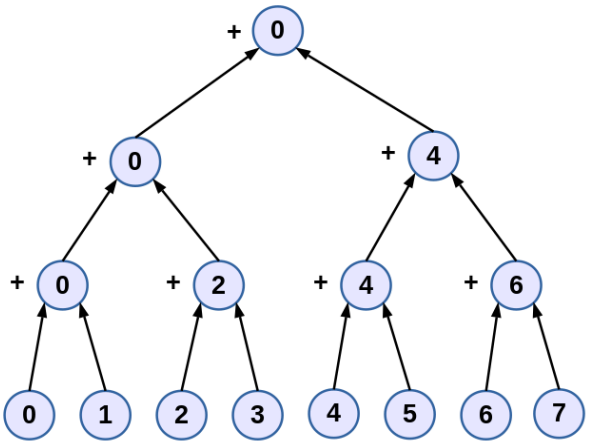

Figura 4.5: Redução

A soma global que estamos tentando calcular é um exemplo de uma classe geral de operações de comunicação coletivas chamada *operações de redução*. Em uma operação global de redução, todos os processos em um comunicador contribuem com dados que são combinados em operações binárias, como exemplificado na Figura 4.5 e 4.6.

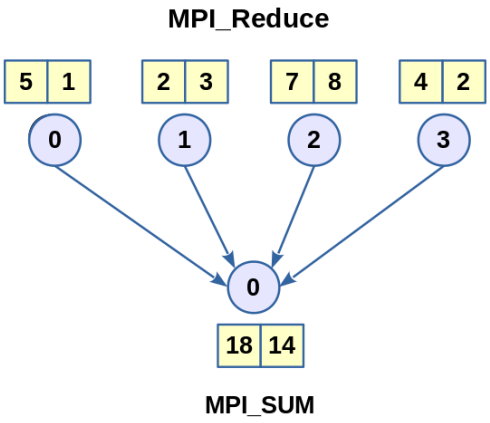

Figura 4.6: Operação de redução

A função **MPI_Reduce** combina todos os elementos presentes no vetor de envio de todos os processos do comunicador usando a operação definida como parâmetro e coloca o valor resultante no vetor de recepção do processo especificado. Operações binárias típicas são adição, máximo, mínimo, e lógico etc. É possível definir

operações adicionais além daquelas definidas por padrão para a função **MPI_Reduce**. A sua sintaxe é a seguinte:

int MPI_Reduce(void* vet_envia, void* vet_recebe, int cont, MPI_Datatype tipo_mpi, MPI_Op oper, int raiz, MPI_Comm com)

A operação **MPI_Reduce** combina os valores armazenados em *vet_envia* usando a operação *oper* e armazena o resultado em *vet_recebe* no processo *raiz*. Tanto *vet_envia* como *vet_recebe* são vetores com dimensão igual ao parâmetro *cont*, indicando o total de posições de memória contíguas com o tipo de dados *tipo_mpi*.

A função **MPI_Reduce** deve ser chamada por todos os processos no comunicador *com*. Os valores de *cont*, *tipo_mpi* e *oper* devem ser idênticos em cada processo. O argumento *oper* pode ter qualquer um dos valores predefinidos na Tabela 4.1. Uma especial atenção merece ser dada aos operadores **MPI_MINLOC** e **MPI_MAXLOC**, veja maiores detalhes sobre o uso desses operadores ao final desta seção.

Tabela 4.1: Operações de redução e seu significado

Nome da Operação	**Significado**
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Soma
MPI_PROD	Produto
MPI_LAND	"E" lógico
MPI_BAND	"E" bit a bit
MPI_LOR	"Ou" lógico
MPI_BOR	"Ou" bit a bit
MPI_LXOR	"Ou Exclusivo" lógico
MPI_BXOR	"Ou Exclusivo" bit a bit

Nome da Operação	**Significado**
MPI_MAXLOC	Máximo e posição*
MPI_MINLOC	Mínimo e posição*

Note que cada processo chama a rotina **MPI_Reduce** com os mesmos argumentos. Em particular, embora a variável *cont* tenha apenas significado no processo *raiz*, cada processo deve fornecê-lo como argumento. Podemos ver a utilização da função **MPI_Reduce** no seguinte exemplo:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define TAM 5
int main(int argc, char *argv[]) { /* mpi_reduce.c  */
int meu_ranque, num_procs, i, raiz = 0;
float vet_envia [TAM] ; /* Vetor a ser enviado  */
float vet_recebe [TAM]; /* Vetor a ser recebido */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Preenche o vetor com valores que dependem do ranque */
    for (i = 0; i < TAM; i++)
        vet_envia[i] = (float) (meu_ranque*TAM+i);
    /* Faz a redução, encontrando o valor máximo do vetor */
     MPI_Reduce(vet_envia, vet_recebe, TAM, MPI_FLOAT, MPI_MAX,
raiz, MPI_COMM_WORLD);
    /* O processo raiz imprime o resultado da redução */
    if (meu_ranque == raiz) {
        for (i = 0; i < TAM; i++)
            printf("vet_recebe[%d] = %3.1f ", i,vet_recebe[i]);
        printf("\n\n");     }
    MPI_Finalize();
    return(0);
}
```

O programa apresentado preenche um vetor *vet_envia* com valores diferentes em cada elemento, de acordo com o ranque do processo e o índice Math input error do elemento. Depois, é feita uma redução para encontrar o maior valor para cada posição

Math input error do vetor em todos dos processos. O resultado da redução é recebido no vetor *vet_recebe* no processo com ranque igual a 0. Um possível resultado para a execução desse programa com 4 processos seria:

```
vet_recebe[0]    =    15.000000    vet_recebe[1]    =    16.000000
vet_recebe[2] = 17.000000
vet_recebe[3] = 18.000000 vet_recebe[4] = 19.000000
```

Finalmente, podemos reescrever as últimas linhas do programa do método do trapézio da Seção 3.8, substituindo o laço pela rotina de redução, que é uma forma muito mais eficiente para o cálculo do valor final da integral:

```
. . .
/* Soma as integrais calculadas por cada processo */
MPI_Reduce(&integral,   &total,   1,   MPI_FLOAT,   MPI_SUM,   0,
MPI_COMM_WORLD);
/* Imprime o resultado */
. . .
```

Operadores de redução MPI_MAXLOC e MPI_MINLOC

Na Tabela 4.1 os operadores **MPI_MINLOC** e **MPI_MAXLOC** foram apresentados, mas sem grandes detalhes. Vamos estudar agora esses operadores com mais atenção.

O operador **MPI_MINLOC** é usado para computar o mínimo global e também um índice associado a esse valor mínimo. O operador **MPI_MAXLOC** similarmente computa um máximo global e o seu respectivo índice associado. Ambas as operações são associativas e comutativas. Para que essas operações sejam computadas corretamente, é necessário passar um tipo MPI especial como parâmetro de entrada contendo um par: valor e índice. Se esse par contiver, por exemplo, um valor e o ranque do processo, esses operadores poderão ser utilizados para computar um mínimo (máximo) e o ranque do processo contendo esse valor.

Tabela 4.2: Tipos especiais MPI_MAXLOC e MPI_MINLOC

Nome	**Descrição**
MPI_FLOAT_INT	float e int
MPI_DOUBLE_INT	double e int
MPI_LONG_INT	long e int
MPI_2INT	par de int
MPI_SHORT_INT	short e int
MPI_LONG_DOUBLE_INT	long double e int

Em C, o tipo especial fornecido pelo MPI contém um par que pode ter tipos diferentes, onde o índice é sempre um inteiro. O MPI oferece vários tipos especiais predefinidos para uso com os operadores **MPI_MAXLOC** e **MPI_MINLOC**. Na Tabela 4.2 encontramos os tipos disponíveis para a linguagem C.

No exemplo seguinte, cada processo tem um vetor de 10 *doubles* em C. Para cada uma das 10 posições, o exemplo calcula o valor e o ranque do processo contendo o maior valor.[2]

```
#include <stdio.h>
#include "mpi.h"
#define TAM 10
int main(int argc, char *argv[]) { /* mpi_maxloc.c  */
int meu_ranque, i, raiz = 0, indice[10];
double                    vetor_entrada[TAM]                    =
{0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};
double  vetor_saida[TAM];
struct {
     double valor;
     int   ranque;
} entrada[10], saida[10];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
     /* Só os processos com ranques pares colocam valor 50 no
vetor */
    if ((meu_ranque%2) == 0)
```

```
        vetor_entrada[meu_ranque] = 50.0;
    /* Prepara os parâmetros de entrada função de redução */
    for (i = 0; i < TAM; ++i) {
         entrada[i].valor = vetor_entrada[i];
         entrada[i].ranque = meu_ranque;
    }
    /* Faz a operação de redução com o tipo MPI_DOUBLE_INT */
    MPI_Reduce(entrada, saida, TAM, MPI_DOUBLE_INT, MPI_MAXLOC,
raiz, MPI_COMM_WORLD);
    /* O processo raiz imprime o resultado da redução */
    if (meu_ranque == raiz) {
        for (i = 0; i < TAM; ++i) {
             vetor_saida[i] = saida[i].valor;
             indice[i] = saida[i].ranque;
                 printf("Resultado = %3.1f Processo = %d \n",
vetor_saida[i], indice[i]);
        }
    }
    /* Termina a execução */
    MPI_Finalize();
    return(0);
}
```

Um resultado possível para a execução desse exemplo com 10 processos seria:

```
Resultado = 50.0 Processo = 0
Resultado = 1.0 Processo = 0
Resultado = 50.0 Processo = 2
Resultado = 3.0 Processo = 0
Resultado = 50.0 Processo = 4
Resultado = 5.0 Processo = 0
Resultado = 50.0 Processo = 6
Resultado = 7.0 Processo = 0
Resultado = 50.0 Processo = 8
Resultado = 9.0 Processo = 0
```

4.6 Redução com difusão

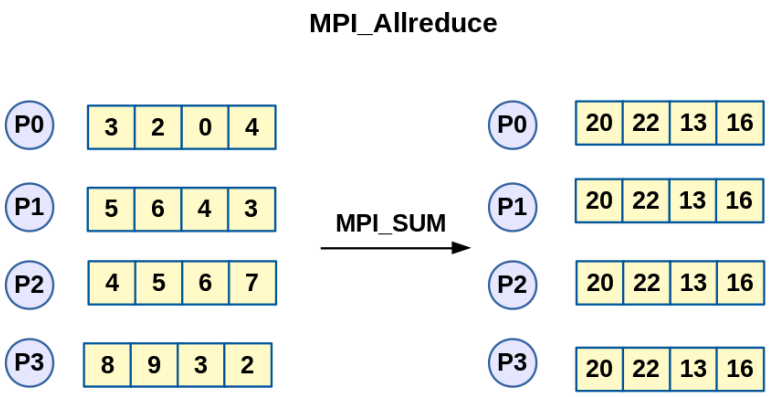

Figura 4.7: Operação de redução com difusão

A função **MPI_Allreduce** permite realizar a operação de redução com difusão, cujo protótipo vemos aqui:

int MPI_Allreduce (void* vet_envia, void* vet_recebe, int cont, MPI_Datatype tipo_mpi, MPI_Op oper, MPI_Comm com)

Os seus parâmetros são similares aos da função **MPI_Reduce**, com a diferença de que não existe o parâmetro *raiz*, já que o resultado da operação *oper* será recebido por todos os processos do comunicador em *vet_recebe*. Vejamos no exemplo seguinte o uso da função **MPI_Allreduce** para calcular a média dos elementos de um vetor.[3]

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#define NELEM 1024
int main(int argc, char *argv[]) { /* mpi_allreduce.c  */
int i, meu_ranque, num_procs;
float *nums_aleat = NULL;
float media, dif_quad_local = 0, soma_local = 0;
float soma_global, dif_quad_global, desvio_padrao;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

```
   /* Sementes diferentes para cada processo  */
   srand(meu_ranque+meu_ranque*meu_ranque);
   /* Cria um vetor de números aleatórios em cada processo. */
   nums_aleat  = (float *)malloc(sizeof(float) * NELEM);
   for (i = 0; i < NELEM; i++)
        nums_aleat[i] = (rand() / (float)RAND_MAX);
   /* Soma os números localmente */
   for (i = 0; i < NELEM; i++)
        soma_local += nums_aleat[i];
   /* Imprime a soma e média dos números em cada processo */
    printf("Soma local para o processo %d - %f, media local =
%f\n", meu_ranque, soma_local, soma_local / NELEM);
   /* Faz redução das somas locais para calcular a média global
*/
      MPI_Allreduce(&soma_local,  &soma_global,  1,  MPI_FLOAT,
MPI_SUM, MPI_COMM_WORLD);
   media = soma_global / (NELEM * num_procs);
    /* Computa a soma local do quadrado das diferenças da média
*/
   for (i = 0; i < NELEM; i++)
   dif_quad_local += (nums_aleat[i] - media) * (nums_aleat[i] -
media);
    /* Faz a redução da soma global do quadrado das diferenças
locais */
        MPI_Allreduce(&dif_quad_local,  &dif_quad_global,  1,
MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    /* Calcula o desvio padrão com a média do quadrado das
diferenças */
             desvio_padrao  =  sqrt(dif_quad_global/(NELEM   *
num_procs));
         printf("Média - %f, Desvio padrão  = %f Meu ranque=
%d\n", media, desvio_padrao, meu_ranque);
   /* Libera espaço do vetor */
   free(nums_aleat);
   MPI_Finalize();
   return(0);
}
```

Nesse exemplo, cada processo cria um vetor local de números aleatórios e calcula a soma local dos seus elementos. Cada processo chama a função **MPI_Allreduce** para totalizar as somas locais em uma soma global, que é difundida para todos processos. Cada

processo calcula a média global usando o número elementos do vetor local e o total de processos do comunicador *num_procs*.

Por sua vez, essa média global será então utilizada em cada processo para calcular a soma local do quadrado das diferenças. Uma vez que essa soma local seja computada, a soma global do quadrado das diferenças é calculada com o uso da função **MPI_Reduce**.

Finalmente, a soma global do quadrado das diferenças é utilizada pelo processo *raiz* para computar e imprimir o valor do desvio padrão. Um possível resultado da execução desse programa com 4 processos seria:

```
Soma local para o processo 2 - 509.407440, media local =
0.497468
Soma local para o processo 3 - 493.486481, media local =
0.481920
Soma local para o processo 1 - 531.957336, media local =
0.519490
Soma local para o processo 0 - 518.912537, media local =
0.506751
Média - 0.501407, Desvio padrão  = 0.287490 Meu ranque= 2
Média - 0.501407, Desvio padrão  = 0.287490 Meu ranque= 0
Média - 0.501407, Desvio padrão  = 0.287490 Meu ranque= 1
Média - 0.501407, Desvio padrão  = 0.287490 Meu ranque= 3
```

4.7 Coleta com difusão

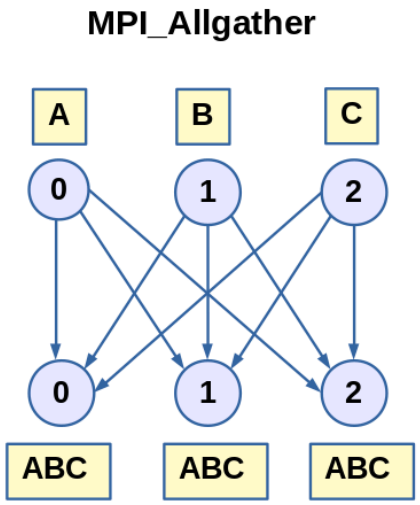

Figura 4.8: Operação de coleta com difusão

Essa variante da operação de coleta pode também ser entendida como uma combinação de coleta com difusão, onde todos os processos, e não apenas o processo **raiz**, recebem a mensagem. A sintaxe da função **MPI_Allgather** é a seguinte:

int MPI_Allgather(void* vet_envia, int cont_envia, MPI_Datatype tipo_envia, void* vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, MPI_comm com)

A função **MPI_Allgather** junta o conteúdo de *vet_envia* em cada processo. Seu efeito é equivalente a uma sequência de Math input error chamadas a **MPI_Gather**, cada qual com um processo diferente agindo como processo **raiz**. O próximo exemplo apresenta um uso prático da função **MPI_Allgather** em um programa para calcular a média dos elementos de um vetor.[4]

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#define NELEM 1024
int main(int argc, char *argv[]) {  /* mpi_allgather.c  */
int i, meu_ranque, num_procs;
```

```
float *medias_parc = NULL;
float *sub_nums_aleat = NULL;
float soma_local=0, soma_medias=0;
float media, media_local ;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   /* Sementes diferentes para cada processo  */
   srand(meu_ranque+meu_ranque*meu_ranque);
   /* Cria um vetor de números aleatórios em cada processo */
   sub_nums_aleat  = (float *)malloc(sizeof(float) * NELEM);
   for (i = 0; i < NELEM; i++)
        sub_nums_aleat[i] = (rand() / (float)RAND_MAX);
   /* Soma os números localmente */
   for (i = 0; i < NELEM; i++)
        soma_local += sub_nums_aleat[i];
   media_local = soma_local / NELEM;
   /* Imprime a soma e média dos números em cada processo */
    printf("Soma local para o processo %d - %f, media local =
%f\n", meu_ranque, soma_local, media_local);
   /* Coleta todas as médias parciais de todos os processos */
   medias_parc = (float*)malloc(sizeof(float) * num_procs);
     MPI_Allgather(&media_local, 1, MPI_FLOAT, medias_parc, 1,
MPI_FLOAT, MPI_COMM_WORLD);
   /* Calcula a média total de todos os números. */
   for (i = 0; i < num_procs; i++)
        soma_medias += medias_parc[i];
   media = soma_medias / num_procs;
   printf("Média de todos os elementos calculada no processo %d
é  %f\n", meu_ranque, media);
   /* Libera espaço alocado  */
   free(medias_parc);
   free(sub_nums_aleat);
   MPI_Finalize();
   return(0);
}
```

Vemos em seguida uma saída possível para a execução desse programa com 4 processos:

```
Soma local para o processo 3 - 499.603455, media local =
0.487894
Soma local para o processo 0 - 518.912537, media local =
```

```
0.506751
Soma local para o processo 1 - 511.486267, media local =
0.499498
Soma local para o processo 2 - 513.696533, media local =
0.501657
Média de todos os elementos calculada no processo 2 é  0.498950
Média de todos os elementos calculada no processo 3 é  0.498950
Média de todos os elementos calculada no processo 1 é  0.498950
Média de todos os elementos calculada no processo 0 é  0.498950
```

Como esperado, a média global calculada é a mesma em todos os processos.

4.8 Transposição

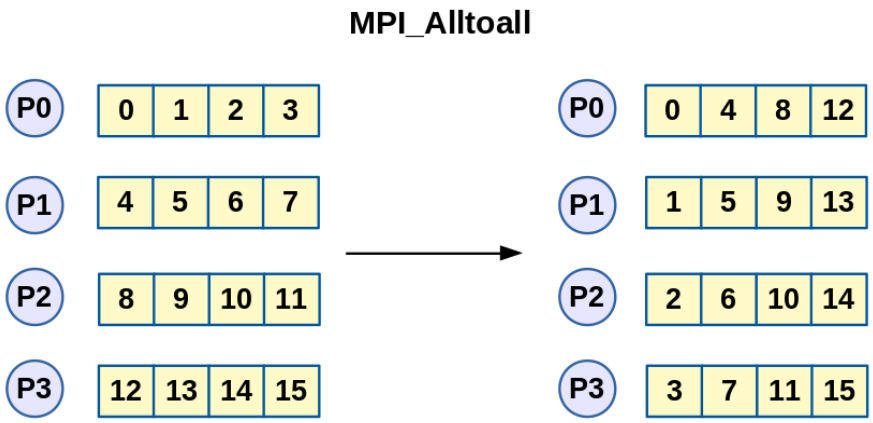

Figura 4.9: Operação de transposição

int MPI_Alltoall(const void *vet_envia, int cont_envia, MPI_Datatype tipo_envia, void *vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, MPI_Comm com)

Essa operação coletiva é uma extensão da operação de coleta com difusão (**MPI_Allgather**), onde cada processo envia um dado diferente para cada um dos receptores. O *j-ésimo* dado enviado pelo processo Math input error é recebido pelo processo Math input error e armazenado no *i-ésimo* bloco do seu *vetor* de recepção.

O parâmetro *conta_envia* representa o número de elementos do tipo *tipo_envia* a serem enviados para cada processo, assim como o parâmetro *cont_recebe* representa o número de elementos do tipo *tipo_recebe* recebidos por cada processo. Os parâmetros *vet_envia* e *vet_recebe* são ponteiros para os vetores de envio e recepção.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_alltoall.c  */
int i, meu_ranque, num_procs;
int *vetor_entrada, *vetor_saida;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Aloca e atribui valor inicial os dados */
    vetor_entrada = (int*) malloc(num_procs * sizeof(int));
    vetor_saida  = (int*) malloc(num_procs * sizeof(int));
    for (i = 0; i < num_procs; i++)
         vetor_entrada[i] = meu_ranque * num_procs + i;
    /* Imprime a entrada dada para a função MPI_Alltoall */
    printf("Entrada do processo %d: ", meu_ranque);
    for (i = 0; i < num_procs; i++)
         printf("%d ", vetor_entrada[i]);
    printf("\n");
    /* Reorganiza os dados */
      MPI_Alltoall(vetor_entrada, 1, MPI_INT, vetor_saida, 1,
MPI_INT, MPI_COMM_WORLD);
    /* Imprime a saida da função MPI_Alltoall */
    printf("Saida do processo %d: ", meu_ranque);
    for (i = 0; i < num_procs; i++)
         printf("%d ", vetor_saida[i]);
    printf("\n");
    /* Desaloca os vetores e finaliza o MPI */
    free(vetor_saida);
    free(vetor_entrada);
    MPI_Finalize();
    return(0);
}
```

O exemplo anterior ilustra a utilização da função **MPI_Alltoall** na prática. Cada processo tem um vetor de entrada cujo comprimento é igual ao número de processos. A função **MPI_Alltoall** rearranja esses valores de forma que o processo Math input error recebe o *k-éssimo* elemento de cada processo.[5]

Poderíamos ter como resultado para execução desse programa com 4 processos a seguinte saída:

```
Entrada do processo 0: 0 1 2 3
Entrada do processo 1: 4 5 6 7
Entrada do processo 2: 8 9 10 11
Entrada do processo 3: 12 13 14 15
Saida do processo 1: 1 5 9 13
Saida do processo 0: 0 4 8 12
Saida do processo 3: 3 7 11 15
Saida do processo 2: 2 6 10 14
```

Nesse código, o número de processos é igual ao número de elementos de cada vetor. Mas isso não precisa ser sempre assim, também é possível enviar mais de um elemento para cada processo. Para isso devemos alterar o valor de *cont_envia* e *cont_recebe* de acordo com o número de elementos a serem enviados e aumentar a dimensão dos vetores de forma correspondente.

4.9 Algumas observações

A seguir, listamos algumas observações relativas ao uso das rotinas de comunicação coletiva do MPI.

Recomendações

- Apenas o processo **raiz** deve fazer operações de entrada e saída, seja com a entrada e saída padrão, seja com arquivos. Os dados podem ser coletados/distribuídos entre os processos com

o uso das rotinas **MPI_Bcast(), MPI_Gather() e MPI_Scatter()**;

- Não se deve tentar receber com **MPI_Recv()** uma mensagem enviada com **MPI_Bcast()**, já que isso configura um erro;

- Uma rotina de comunicação coletiva deve ser invocada por todos os processos do comunicador. Não se deve, portanto, colocar a chamada coletiva protegida por um *if* que dependa do valor do ranque do processo;

- Não se pode assumir que todos os processos que fazem a chamada de uma rotina de comunicação coletiva vão completar ao mesmo tempo. Se for necessário garantir a sincronização utilize a rotina **MPI_Barrier()**;

- Certifique-se sempre de que os *buffers* de envio e recepção das rotinas coletivas sejam distintos um do outro. Nunca utilize os mesmos *buffers* para envio e recepção.

Otimizando o desempenho

- Utilize as rotinas de comunicação coletiva sempre que possível, elas são otimizadas e permitem ganhos significativos de desempenho sobre soluções caseiras.

- A rotina de comunicação **MPI_Alltoall()** é cara, portanto evite seu quando for possível;

- Tenha cuidado com a quantidade de memória utilizada pelo seu processo. Para grandes vetores, por exemplo, aloque apenas a quantidade necessária para as rotinas **MPI_Gather() e MPI_Scatter()**. Reutilize os *buffers* de tamanho grande ou os libere após o uso. Tenha cuidado, claro, com a sobreposição de sua utilização.

4.10 Estudo de caso: multiplicação de matriz por vetor

Apresentaremos detalhes de uma implementação em MPI para a multiplicação de uma matriz por um vetor, qual seja, Math input error, onde A é uma matriz Math input error, Math input error é um vetor de dimensão Math input error, então Math input error é um vetor de dimensão Math input error, conforme visto a seguir.

Math input error = Math input error Math input error = Math input error
Math input error = Ab = Math input error

onde,

Math input error

Esse problema é interessante, pois além da paralelização de trabalho, há necessidade de uso de operações de comunicação coletivas. Vejamos um exemplo.

A Math input error b = c
Math input error Math input error Math input error = Math input error

O valor de Math input error é calculado pelo produto interno (produto escalar) entre a linha Math input error da matriz Math input error e o vetor Math input error, o que demanda Math input error multiplicações e Math input error adições, resultando em uma complexidade computacional de Math input error para o algoritmo sequencial. O valor de Math input error não depende do valor de Math input error, logo há uma grande

possibilidade de exploração de paralelismo. Apresentamos o código da versão sequencial desse algoritmo na listagem a seguir.

```
#include <stdio.h>
#include <stdlib.h>
void mxv(int m, int n, double* A, double* b, double* c)
{
int i, j;
   for (i = 0; i < m; i++) {
       c[i] = 0.0;
       for (j = 0; j < n; j++)
            c[i] += A[i*n + j]*b[j];
   }
}
int main(int argc, char *argv[]) { /* mxv_seq.c  */
double *A,*b,*c;
int i, j, m, n;
    printf("Por favor entre com m e n: ");
    scanf("%d %d",&m,&n);
    printf("\n");
    A=(double *)malloc(m*n*sizeof(double));
    b=(double *)malloc(n*sizeof(double));
    c=(double *)malloc(m*sizeof(double));
     printf("Atribuindo valor inicial à matriz A e ao vetor
b\n");
   for (j = 0; j < n; j++)
        b[j] = 2.0;
   for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
             A[i*n + j] = i;
   printf("Mutiplicando a matriz A com o vetor b\n");
   (void) mxv(m, n, A, b, c);
   free(A);
   free(b);
   free(c);
   return(0);
}
```

No exemplo em questão, o acesso à matriz Math input error foi linearizado, ou seja, um arranjo bidimensional foi transformado em um vetor. Essa é uma técnica comumente empregada para otimizar

o acesso à matriz, permitindo ainda, se desejado, o acesso com o uso de ponteiros.

A grande dificuldade da paralelização desse programa se encontra em realizar um adequado particionamento de dados entre os diversos processos envolvidos na computação. Um raciocínio mais simples consiste em enviar toda a matriz Math input error e o vetor Math input error para todos os Math input error processos simultaneamente e então cada processo calculará Math input error elementos do vetor Math input error.

Contudo, a grande movimentação de dados envolvida e o seu correspondente alto de custo de comunicação inviabilizam essa opção de distribuição de dados. Algumas alternativas possíveis são as seguintes:

1. Decompor a matriz Math input error em blocos, no sentido das suas linhas, e replicar o vetor Math input error em todos os processos;

2. Decompor a matriz Math input error em blocos, no sentido de suas colunas, decompor os vetores em blocos pelos processos;

3. Decompor a matriz A em blocos de tamanho Math input error, e decompor os vetores em blocos pelos processos.

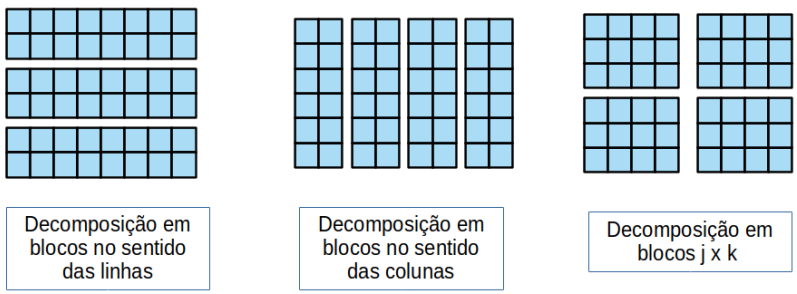

Figura 4.10: Decomposição da Matriz

Cada uma dessas formas de decomposição tem as suas vantagens e desvantagens, além de complexidades distintas. Por simplicidade, vamos assumir que utilizaremos a primeira alternativa de distribuição de dados, com cada processo possuindo um bloco de linhas da matriz Math input error e os vetores Math input error e Math input error replicados em cada processo.

O envio das linhas da matriz Math input error para todos os processos pode ser feito com a rotina **MPI_Scatter** ou ainda com versões simples das rotinas de comunicação ponto a ponto, como **MPI_Send** e **MPI_Recv**. Podemos utilizar a rotina de comunicação coletiva **MPI_Bcast** para difundir o vetor Math input error para todos os processos. Contudo, uma simplificação comum na paralelização é assumir que cada processo possa fazer a leitura ou a geração das linhas da matriz. Isso foi assumido no exemplo apresentado com uma versão simplificada desse algoritmo, onde cada processo gera as suas linhas e a cópia do vetor Math input error para calcular o elemento Math input error. O resultado final vai estar disponível para todos os processos com uso de uma rotina de coleta seguida de difusão. (**MPI_Allgather**).

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
void mxv(long int m, long int n, double* A, double* b, double*
c)
{
long int i, j;
register double temp;
   for (i = 0; i < m; i++) {
        temp = 0.0;
        for (j = 0; j < n; j++)
             temp += A[i*n + j]*b[j];
        c[i] = temp;
   }
}
int main(int argc, char *argv[]) { /* mpi_mxv.c  */
double *Aloc, *b,*cloc, *c;
```

```
/* matriz m x n por um vetor de dimensão n */
long int i, j, m, n;
int meu_ranque, num_procs, raiz=0;
double inicio, fim;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    /* Os processos fazem a leitura da dimensão m e n  como
parâmetros */
   if (argc < 3) {
        if (meu_ranque == 0)
        printf("Entre com as dimensões da matriz: m e n valores
inteiros, sendo m múltiplo do número de processos \n");
        return 0;
    } else {
        m = atoi(argv[1]);
        n = atoi(argv[2]);
    }
    if (meu_ranque == 0){
        printf("Valor de m: %ld e n: %ld \n", m,n);
    }
    Aloc=(double *) malloc(m/num_procs*n*sizeof(double));
    b=(double*) malloc(n*sizeof(double));
    c=(double*) malloc(m*sizeof(double));
    cloc=(double*) malloc(m/num_procs*sizeof(double));

    /* Atribuindo valor inicial à matriz A e ao vetor b */
     for (j = 0; j < n; j++)
              b[j] = 2.0;
     for (i = 0; i < m/num_procs; i++)
          for (j = 0; j < n; j++)
              Aloc[n*i+j] = (double) j;
     inicio = MPI_Wtime();
    /* Cada processo faz o cálculo parcial de 'c'  */
     mxv(m/num_procs, n, Aloc, b, cloc);
     MPI_Barrier(MPI_COMM_WORLD);
     /* O processo raiz coleta o vetor 'c' dos demais processos
*/
            MPI_Allgather(cloc,  m/num_procs,  MPI_DOUBLE,  c,
m/num_procs, MPI_DOUBLE, MPI_COMM_WORLD);
     fim = MPI_Wtime();
```

```
    /* O processo raiz imprime o resultado */
     if (meu_ranque == raiz) {
        printf("Tempo total = %f\n", fim-inicio);
     }
     free(Aloc);
     free(b);
     free(c);
     free(cloc);

     MPI_Finalize();
     return(0);
}
```

O processo **raiz** faz a coleta e a difusão de todos os elementos do vetor Math input error calculados em paralelo. Cada conjunto de Math input error linhas será processada por um processo distinto e o número de colunas Math input error da matriz e do vetor Math input error é um parâmetro de entrada. Como dito, as matrizes locais e o vetor Math input error têm os valores iniciais gerados por cada processo 0.

Avaliação de desempenho

Uma análise simples de complexidade, supondo-se Math input error, indica uma complexidade computacional sequencial de Math input error. Quando Math input error processos são utilizados, cada processo é responsável por Math input error linhas de A, resultando em uma complexidade computacional por processo, sem custos de comunicação, igual a Math input error.

Para os custos de comunicação, não incluindo o custo do envio da linha Math input error e do vetor Math input error, e apenas a coleta e difusão do vetor de resultado Math input error para todos os processos, temos a seguinte situação. Um algoritmo eficiente para a função **MPI_Allgather** requer que cada processo envie Math input error mensagens, com o número total de elementos

enviados por processo igual a Math input error. Então a complexidade de comunicação é igual a Math input error, e a complexidade total desse algoritmo igual a Math input error(J et al. 2007).

Nos experimentos vamos utilizar um computador com uma arquitetura de multiprocessamento simétrico e um total de 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz) para avaliação de desempenho.

O programa deste estudo de caso deve ser salvo com o nome *mpi_mxv.c*. Em seguida, deve ser compilado e executado com os seguintes comandos em um terminal de um sistema operacional do tipo Linux:

```
$ mpicc -o mxv mpi_mxv.c -O3
$ mpirun -np 4 ./mxv 90000 90000
```

O resultado da execução desse programa será o seguinte:

```
Valor de m: 90000 e n: 90000
Tempo total = 2.515200
```

A medição de tempo foi feita tanto na solução sequencial quanto na solução paralela. O tempo medido para a execução da solução sequencial com Math input error e Math input error foi de 10 segundos. As medições e os cálculos de *speedup* e eficiência para as demais medições podem ser vistos na Tabela 4.3.

Tabela 4.3: Métricas da multiplicação matriz-vetor - Amdahl

Processos	**Tempo Total**	***Speedup***	**Eficiência**
1	10 s	1,0	1,00
2	5 s	1,9	0,96
4	3 s	3,8	0,95
8	1 s	6,9	0,86

Processos	Tempo Total	*Speedup*	Eficiência
12	1 s	9,7	0,81
16	1 s	12,7	0,79
20	1 s	14,8	0,74
24	1 s	14,7	0,61

A Figura 4.11 mostra a evolução do *speedup* de acordo com o aumento de processos, de acordo com os dados obtidos a partir da Tabela 4.3. A linha contínua nesse gráfico indica o *speedup* linear ideal e a outra linha com pontos destacados apresenta o resultado do *speedup* calculado. Percebe-se que há uma forte tendência de estabilização do ganho a partir da utilização de 20 processos, quando a curva se torna em uma reta.

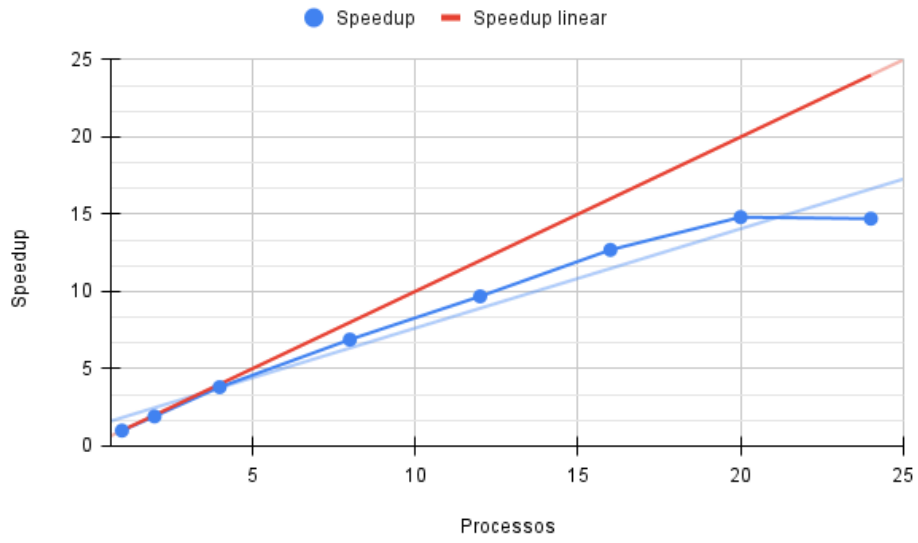

Figura 4.11: Gráfico de *speedup* - multiplicação matriz-vetor - Amdahl

Já a eficiência pode ser observada no gráfico da Figura 4.12 que decai gradativamente a partir de 4 processos. A partir do uso de 12 processos, com um tempo de computação por volta de 1 segundo, a eficiência decai mais fortemente, indicando sua tendência de perda de desempenho. Essa indicação confirma a tendência visualizada na curva de *speedup* apresentada na Figura 4.11.

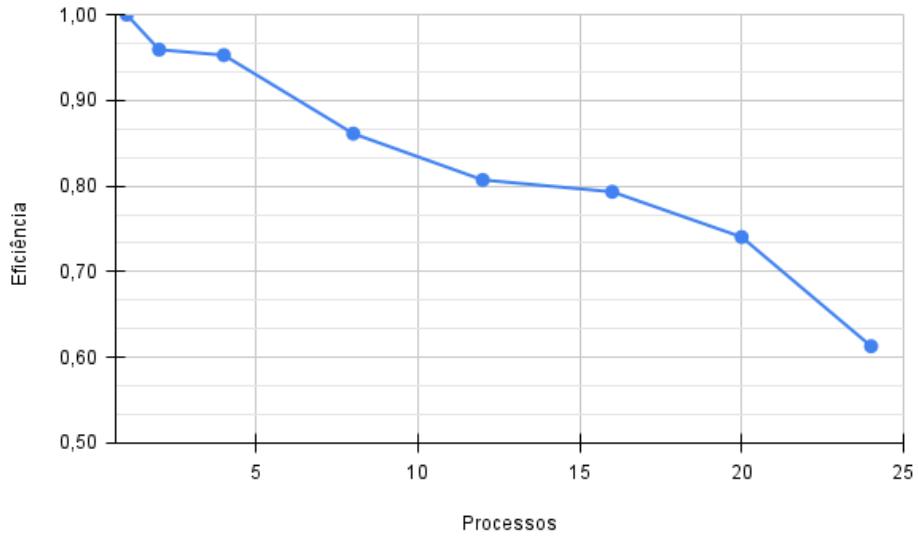

Figura 4.12: Gráfico de eficiência - multiplicação matriz-vetor - Amdahl

4.11 Exercícios propostos

1. Com relação ao uso das rotinas de comunicação coletivas, responda:

    1. Quais as restrições existentes no uso das operações de comunicação coletivas?

    2. O que é chamado de processo **raiz** nas operações de comunicação coletiva?

    3. Qual a finalidade da função **MPI_Barrier**? Quais os cuidados que se deve ter na sua utilização?

    4. Qual a finalidade da função **MPI_Bcast**? A função **MP_Recv** pode ser utilizada para receber mensagens enviadas pela função **MPI_Bcast**?

    5. A função **MPI_Bcast** pode ser utilizada para a sincronização entre processos? Por quê?

2. Um programa contém uma chamada para a função **MPI_Bcast** com o tipo de dados MPI_DOUBLE usado em cada processo. Qual a relação entre o valor que o argumento *cont* deve ter no processo **raiz** e nos demais processos? Devem ser exatamente iguais? Um pode ser maior que o outro?

3. Modifique o exemplo da Seção 4.2, com a rotina **MPI_Bcast**, para o envio de um vetor de *floats* com tamanho fixo.

4. Explique se e por que o seguinte trecho de código MPI está correto ou não:

   Processo 0 executa:

```
MPI_Bcast(&meu_dado, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

Todos os outros processos:

MPI_Recv(&meu_dado, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD,
&estado);
```

5. Com relação ao programa da Seção 4.3 com um exemplo de uso da função **MPI_Scatter**:

   1. Execute o programa com 2, 4 e 8 processos. Verifique se há modificação na saída do programa.

   2. Modifique o valor do processo **raiz**. Execute o programa e verifique se há alguma alteração na saída.

   3. Modifique o programa para que sejam enviados e recebidos dados do tipo **MPI_FLOAT** gerados aleatoriamente.

   4. Modifique o programa de forma que o processo **raiz** pergunte o tamanho dos dados a serem enviados para cada processo. Em seguida essa informação deve ser enviada com uso da função **MPI_Bcast** para os demais processos.

Todos os vetores locais devem ser alocados dinamicamente a partir dessa informação.

6. Com relação às rotinas de comunicação coletivas a seguir, responda:

    1. Qual o significado do argumento *cont_envia* na rotina **MPI_Scatter**? Ele deve ter um conteúdo válido em todos os processos?

    2. Quais as diferenças entre as funções **MPI_Scatter** e **MPI_Bcast**?

    3. Qual o significado do argumento *cont_recebe* na rotina **MPI_Gather**? Ele deve ter um conteúdo válido em todos os processos?

    4. Na função **MPI_Gather**, qual a ordem em que o processo **raiz** armazena as mensagens recebidas dos processos emissores?

    5. Enumere e descreva quatro tipos de operações de redução utilizadas na função **MPI_Reduce**

7. Com relação ao programa da Seção 4.4 com um exemplo da função **MPI_Gather**:

    1. Execute o programa com 2, 4 e 8 processos. Verifique se há modificação na saída do programa.

    2. Modifique o valor do processo **raiz**. Execute o programa e verifique se há alguma alteração na saída.

    3. Modifique o programa para que sejam enviados e recebidos dados do tipo **MPI_FLOAT** gerados aleatoriamente.

4. Modifique o programa de maneira que o processo **raiz** pergunte o tamanho dos dados a serem coletados de cada processo. Em seguida essa informação deve ser enviada com uso da função **MPI_Bcast** para os demais processos. Todos os vetores locais devem ser alocados dinamicamente a partir dessa informação.

8. Altere o código do programa da Seção 4.5, com um exemplo da função **MPI_Reduce**, mudando a operação de redução para MPI_MIN, MPI_SUM e MPI_PROD. Execute em cada um dos casos e verifique as diferenças.

9. Modifique o programa do método do trapézio da Seção 3.8, substituindo o laço final pela rotina de redução. Execute o programa na versão original e modificada com 1, 2, 4 e 8 processos e verifique se há diferenças no tempo de execução.

10. Altere o código do programa da Seção 4.5, com o exemplo com o tipo de dados **MPI_MAXLOC**, mudando o tipo de dados para **MPI_FLOAT_INT** e a operação de redução para **MPI_MINLOC**. Execute e verifique as diferenças.

11. Reescreva o código da Seção 4.6, com um exemplo de uso da função **MPI_Allreduce**, sem o uso dessa função. Execute o código original e o modificado com Math input error e com 2, 4, 6, e 8 processos. Prepare um gráfico comparando os tempos de execução em cada caso.

12. Execute o programa da Seção 4.7, com um exemplo de uso da função **MPI_Allgather**, com 2, 4, 6 e 8 processos. Verifique qual a diferença do resultado na saída do programa.

13. Modifique o código da Seção 4.7, com um exemplo da função **MPI_Allgather**, para que cada processo gere vetores com tamanhos diferentes e esse tamanho seja enviado junto da

média local para os demais processos com uso da função MPI_Allgather.

14. Execute o programa da Seção 4.8, com um exemplo da função **MPI_Alltoall**, com 2, 4, 6 e 8 processos. Verifique qual a diferença do resultado na saída do programa.

15. Modifique o programa da Seção 4.8, com um exemplo da função **MPI_Alltoall**, para que cada processo envie e receba 2 elementos dos demais processos. Quais as diferenças no resultado?

16. Modifique o programa de multiplicação de matriz por vetor, na Seção 4.10, realizando os experimentos para avaliação de desempenho consoante com a teoria de Gustafson. Lembre-se de garantir a carga de trabalho proporcional entre os Math input error processos utilizados no seu experimento.

17. Modifique o programa de multiplicação de matriz por vetor, na Seção 4.10, realizando a divisão para cada processo por colunas, em vez de linhas.

18. Quais são os erros no programa a seguir?

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
int meu_ranque, num_procs;
int k, envia[4], recebe[3];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    if (num_procs != 4) {
        printf("Erro:# de processos deve ser igual 4\n");
        printf("Abortando programa ...\n");
```

```
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    for (k = 0; k < num_procs; k++)
         envia[k] = (k+1) + meu_ranque*num_procs;

    printf("%d : envia = %d %d %d %d\n", meu_ranque, envia[0],
envia[1], envia[2], envia[3]);

      MPI_Alltoall(envia, 2, MPI_FLOAT, recebe, 1, MPI_INT,
MPI_COMM_WORLD);

       printf("%d : recebe = %d %d %d %d\n", meu_ranque,
recebe[0], recebe[1], recebe[2], recebe[3]);

    MPI_Finalize();
    return(0);
}
```

19. Escreva uma rotina para calcular o produto escalar de dois vetores, utilizando rotinas de comunicação coletiva do MPI para envio do vetor inicial e recepção dos resultados parciais pelo processo **raiz**. Considere cada vetor com Math input error posições e divida a operação entre Math input error processos distintos. Considere que a divisão de Math input error por Math input error não tem resto. Assuma que o valor de Math input error NÃO é conhecido e deve ser enviado pelo processo **raiz** para os demais processos com uso de uma rotina comunicação coletiva.

20. Execute o programa do exercício anterior com N = 100.000.000 e com 1, 2, 4, 6 e 8 processos em uma máquina com 8 processadores. Instrumente o código com uso da rotina **MPI_Wtime** e trace os gráficos com as curvas de tempo de execução, *speedup* e eficiência em função do número de processos.

1. A partir da especificação 3.0 do MPI, foram criadas versões não bloqueantes das operações de comunicação coletivas, mas não serão abordadas neste livro↩

2. Adaptado de http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node79.html↩

3. Adaptado de http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/↩

4. Adaptado de http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/↩

5. Adaptado de http://www.codeproject.com/Articles/896437/A-Gentle-Introduction-to-the-Message-Passing-Interface↩

5 Comunicação MPI em detalhes

O MPI tem diversas possibilidades para a comunicação entre dois processos. Primeiramente, as rotinas de envio e recepção podem ser categorizadas como bloqueantes ou não bloqueantes. Adicionalmente, o envio e recepção de mensagens podem ser realizados de quatro modos distintos: síncrono, pronto, *bufferizado* ou padrão. Essas diversas possibilidades, juntamente a uma análise de como evitar a ocorrência de *deadlock*, serão discutidas em detalhes a seguir.

5.1 Introdução

As operações de comunicação ponto a ponto do MPI tipicamente envolvem a transmissão de mensagens entre dois, e apenas dois, processos MPI. Enquanto um dos processos realiza a operação de envio, a outra tarefa executa a operação de recepção correspondente. Contudo, existem diversos tipos de rotinas de envio e recepção, utilizadas com propósitos diferentes, e divididas basicamente em dois grandes grupos: bloqueantes e não bloqueantes. Note que as mensagens podem ser recebidas com qualquer tipo de rotina de recepção, independentemente do tipo de rotina utilizada para o envio.

Tabela 5.1: Rotinas de envio e recepção

Modo de Comunicação	**Rotinas Bloqueantes**	**Rotinas Não Bloqueantes**
Síncrono	MPI_Ssend	MPI_Issend
Pronto	MPI_Rsend	MPI_Irsend
Bufferizado	MPI_Bsend	MPI_Ibsend
Padrão	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv

A Tabela 5.1 mostra todas as opções do MPI para as rotinas de envio e recepção de mensagens. A maioria delas é analisada aqui, e a diferença de comportamento entre elas e exemplos de uso são apresentados. Em uma situação ideal, cada operação de envio seria perfeitamente sincronizada com a sua correspondente operação de recepção, mas isso raramente acontece. De uma forma ou de outra, a implementação MPI deve ser capaz de lidar com o armazenamento de dados quando as duas tarefas estão fora de sincronismo. Considere os seguintes casos:

- A operação de envio ocorre alguns segundos antes de a recepção estar pronta - onde fica a mensagem enquanto a recepção está pendente?
- Múltiplos envios chegam à mesma tarefa de recepção, que só pode aceitar um envio de cada vez - o que acontece com as mensagens que estão esperando?

A implementação MPI, e não o padrão MPI, decide o que acontece com os dados nesses tipos de casos. Normalmente, uma área de dados do sistema é reservada para armazenar dados em trânsito. Esse espaço do *buffer* do sistema é:

- Invisível para o programador e gerido inteiramente pela biblioteca MPI;
- Um recurso finito que pode ser esgotado facilmente;

- Muitas vezes misterioso e não bem documentado;
- Capaz de existir no lado de envio, no lado de recebimento, ou em ambos;
- Algo que pode melhorar o desempenho do programa, pois permite que as operações de envio e recepção possam ser assíncronas.

O MPI prevê rotinas (**MPI_Buffer_attach** por exemplo) para o usuário gerenciar explicitamente o *buffer* de envio, evitando comportamentos inesperados do programa, quando o *buffer* de sistema é esgotado.

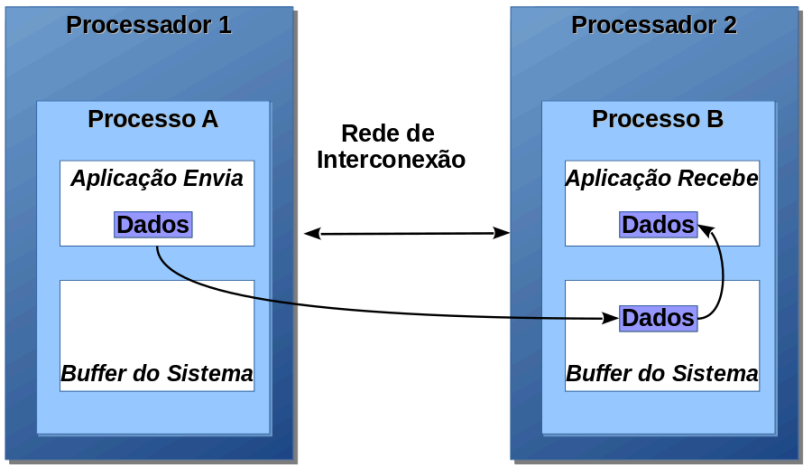

Figura 5.1: Envio e recepção "bufferizados"

O MPI fornece também diversas rotinas associadas com as operações de envio/recepção, como as rotinas utilizadas para esperar a chegada de uma mensagem ou verificar se uma mensagem já foi enviada. Vejamos agora algumas considerações sobre os diversos tipos de rotinas de envio e recepção.

5.2 Rotinas de envio e recepção bloqueantes

Uma rotina de envio bloqueante só vai retornar depois que houver garantia que os dados da aplicação (os seus dados de envio) possam ser reutilizados. Por garantia, entenda-se que modificações posteriores não afetarão os dados que estão sendo enviados. Essa garantia não implica que os dados foram efetivamente recebidos por outro processo - podem muito bem estar situados em um *buffer* do sistema.

Um envio bloqueante pode ser síncrono, obrigando a realização de um protocolo de confirmação com a rotina de recepção, para assegurar o envio completo da mensagem. Um envio bloqueante também pode ser assíncrono, desde que um *buffer* de sistema seja utilizado para armazenar os dados antes de sua distribuição para a rotina de recepção.

O **MPI_Send** não termina enquanto as variáveis de envio não forem esvaziadas (ou seja, as variáveis da aplicação estão disponíveis para reuso). Nesse caso, o sucesso das operações de comunicação depende do tamanho da mensagem e do tamanho do *buffer* do sistema, sendo que as mensagens maiores que o espaço disponível no *buffer* do sistema serão enviadas no modo síncrono.

Uma recepção bloqueante só vai retornar depois que os dados já chegaram e estão prontos para serem utilizados pelo programa. No modo de comunicação bloqueante o **MPI_Recv** não termina enquanto as variáveis de recepção não forem preenchidas (ou seja, quando a mensagem estiver disponível para uso).

Um programa correto em MPI não pode depender do uso de um *buffer* de sistema. Programas assim são chamados **inseguros**, embora possam executar e produzir resultados corretos na maior parte das vezes.

5.3 Rotinas de envio e recepção não bloqueantes

Para garantir que não aconteça *deadlock*, devemos optar pelas rotinas de envio/recepção na versão não bloqueante (veja a Seção 5.6). Com o uso da versão bloqueante, o programa fica bloqueado (parado), esperando até o momento no qual as variáveis utilizadas para o envio da mensagem possam ser reutilizadas com segurança. Na versão não bloqueante, a computação prossegue e, quando for necessário, verificamos se a operação já terminou ou não. Rotinas não bloqueantes permitem superposição da comunicação com a computação, oferecendo melhor desempenho para as aplicações.

As rotinas de envio e recepção não bloqueantes têm um comportamento similar - ambas retornam quase imediatamente. Elas não esperam todos os eventos de comunicação se completarem, tais como a cópia da mensagem da memória do usuário para espaço de *buffer* do sistema ou a chegada real da mensagem.

Operações não bloqueantes simplesmente "pedem" à biblioteca MPI para executar a operação quando for possível. O usuário não pode prever quando isso vai acontecer. Não podemos modificar/utilizar as variáveis da aplicação utilizadas para o envio/recepção da mensagem enquanto não houver certeza de que a operação não bloqueante solicitada foi efetivamente realizada pela biblioteca.

Comunicações não bloqueantes são usadas principalmente para a sobreposição da computação com a comunicação e exploração dos possíveis ganhos de desempenho. Sendo assim, se a programação for feita de maneira adequada, podemos realizar computação e comunicação em paralelo, melhorando o desempenho final do programa.

As operações não bloqueantes, de forma diferente das operações bloqueantes, retornam (imediatamente) um manipulador do tipo **MPI_Request** (*pedido*). Esse manipulador pode ser testado uma única vez (**MPI_Test**) e retornar, ou o processo pode ficar em espera ocupada (**MPI_Wait**) aguardando pela chegada real da mensagem.

Vemos a seguir o protótipo das rotinas de envio e recepção não bloqueantes no **modo padrão** do MPI.

int MPI_Isend(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino, int etiq, MPI_Comm com, MPI_Request *pedido)

int MPI_Irecv(void* mensagem, int cont, MPI_Datatype tipo_mpi, int origem, int etiq, MPI_Comm com, MPI_Request *pedido)

5.4 Esperando a mensagem

Após o uso das rotinas de recepção não bloqueantes, é necessário verificar se a mensagem desejada já chegou efetivamente. Isso é feito basicamente de duas formas: uma rotina que fica em espera indefinida até a chegada da mensagem (**MPI_Wait**) ou podemos apenas testar e retornar imediatamente, indicando se a mensagem já chegou ou não (**MPI_Test**). O protótipo das duas rotinas é visto a seguir. Preste especial atenção ao manipulador *estado* retornado por essas funções, utilizado para identificar a origem e a etiqueta da mensagem recebida.

int MPI_Wait(MPI_Request *pedido, MPI_Status *estado)

int MPI_Test(MPI_Request *pedido, int *flag, MPI_Status *estado)

Algumas vezes é desejável esperar por múltiplas mensagens, quando então podemos passar um vetor de *pedidos* e esperar por todas as mensagens correspondentes (**MPI_Waitall**), ou apenas uma das mensagens (**MPI_Waitany**), ou uma ou mais mensagens (**MPI_Waitsome**):

int MPI_Waitall(int cont, MPI_Request vetor_de_pedidos[], MPI_Status vetor_de_estados[])

int MPI_Waitany(int cont, MPI_Request vetor_de_pedidos[], int *indice, MPI_Status *estado)

int MPI_Waitsome(int cont_entra, MPI_Request vetor_de_pedidos[], int *cont_saida, int vetor_de_indices[], MPI_Status vetor_de_estados[])

A rotina **MPI_Waitsome** apenas garante que pelo menos uma mensagem relativa a um *pedido* foi recebida; não há garantia de que as mensagens relativas a todos os *pedidos* completaram ou que as entradas em *vetor_de_indices* estarão em ordem crescente. Também, as mensagens que foram completadas enquanto a rotina **MPI_Waitsome** estiver sendo executada podem ou não podem ser retornadas como completadas pela rotina. Existem versões correspondentes de **MPI_Test** para cada uma das funções apresentadas. Vejamos os protótipos de cada uma dessas funções.

int MPI_Testall(int cont, MPI_Request vetor_de_pedidos[], int *flag, MPI_Status vetor_de_estados[])

int MPI_Testany(int cont, MPI_Request vetor_de_pedidos[], int *indice, int *flag, MPI_Status *estado)

int MPI_Testsome(int cont_entra, MPI_Request vetor_de_pedidos[], int *cont_saida, int vetor_de_indices[], MPI_Status vetor_de_estados[])

O exemplo seguinte apresenta uma operação de redução com difusão (*allreduce*) utilizando operações de envio e recepção não bloqueantes.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_isend.c  */
int i, meu_ranque, num_procs;
```

```
int pot2, destino, meu_valor;
int reducao, recebido, etiq=1, cont=1;
MPI_Status estado;
MPI_Request pedido_envia;
MPI_Request pedido_recebe;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Aborta se número de processos não for potência de 2 */
    pot2 = num_procs;
    while (((pot2 % 2) == 0) && pot2 > 1)
           pot2 /= 2;
    if (pot2 != 1) {
        if (meu_ranque == 0)
            printf("Por favor execute com número de processos potencia de 2 \n");
        MPI_Finalize();
        exit(0);
    }
    /* Cada processo tem um valor diferente para a redução */
    meu_valor = meu_ranque*num_procs;
    reducao = meu_valor;
    /* Calcula o destino no algoritmo de "recursive doubling" */
    for (i = 1; i <= (num_procs/2); i += i) {
        if ((meu_ranque/i)%2 == 0)
            destino = meu_ranque + i;
        else
            destino = meu_ranque-i;
    /* Posta os envios e recepções em qualquer ordem */
        MPI_Isend(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &pedido_envia);
        MPI_Irecv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &pedido_recebe);
    /* A rotina "MPI_Wait" assegura que os dados já foram transmitidos e recebidos */
        MPI_Wait(&pedido_envia, &estado);
        MPI_Wait(&pedido_recebe, &estado);
    /* Realiza operação de redução com os dados recebidos */
        if (recebido > reducao)
            reducao = recebido;
    }
    printf("Meu valor = %d, redução = %d \n", meu_valor, reducao);
    MPI_Finalize();
    return(0);
}
```

O algoritmo da implementação MPICH do MPI, que utiliza uma técnica de *recursive doubling*(Thakur, Rabenseifner, and Gropp 2005), foi utilizado como base para esse exemplo. Por simplificação, vamos assumir que Math input error, o número de processos, é uma potência de 2 e implementar apenas a operação de redução MPI_MAX (obter o máximo entre todos os valores). O algoritmo requer apenas Math input error passos para a realização do algoritmo de redução e a difusão do resultado para todos os nós, conforme exemplificado na Figura 5.2.

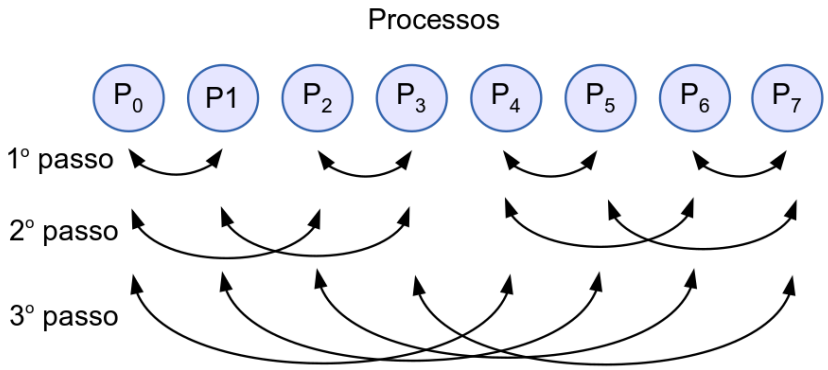

Figura 5.2: Algoritmo "recursive doubling"

O uso de rotinas não bloqueantes é necessário para garantir que não ocorrerá *deadlock*, independente da ordem de chamada das operações de envio e recepção de mensagens ou da "bufferização" das mensagens pelo sistema.

Observamos a seguir o resultado da execução do programa com 8 processos:

```
Meu valor = 24, redução = 56
Meu valor = 56, redução = 56
Meu valor = 8, redução = 56
Meu valor = 16, redução = 56
Meu valor = 32, redução = 56
Meu valor = 40, redução = 56
Meu valor = 48, redução = 56
Meu valor = 0, redução = 56
```

5.5 Modos de comunicação

O MPI define quatro modos de comunicação:

- Modo síncrono (o mais seguro);
- Modo pronto (menor sobrecarga para o sistema);
- Modo "bufferizado" (desacopla o emissor do receptor);
- Modo padrão (solução de compromisso).

O modo de comunicação é selecionado de acordo com a rotina de envio utilizada. Para todos os modos existem versões bloqueantes e não bloqueantes das rotinas de envio, como mostrado na Tabela 5.1. Para uma dada rotina de envio de mensagem, diz-se que foi "postada" uma operação de recepção correspondente, quando um processo qualquer iniciar uma rotina de recepção com comunicador e etiqueta (*tag*) que satisfaçam aos que foram utilizados pela rotina de envio. Note que é permitido o uso de curingas nas operações de recepção tanto para o remetente (**MPI_ANY_SOURCE**) quanto para a etiqueta (**MPI_ANY_TAG**), que devem ser considerados nesse caso. As rotinas **MPI_Recv** e **MPI_Irecv** podem receber mensagens enviadas em qualquer um dos modos de comunicação.

5.5.1 Modo síncrono

No modo síncrono a rotina de envio pode ser iniciada havendo ou não uma rotina de recepção correspondente já postada. Contudo, o envio completará com sucesso apenas quando uma recepção correspondente for postada e a recepção da mensagem enviada pela rotina de envio síncrona for iniciada pela operação de recepção. Esse modo não requer o uso de *buffers* no sistema. Podemos assegurar que um programa está **seguro** se ele executar corretamente utilizando apenas rotinas de envio no modo síncrono. Vejamos a seguir a sintaxe desta função:

int MPI_Ssend(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino, int etiq, MPI_Comm com)

Se tomarmos o exemplo da Seção 5.3 e substituirmos as funções não bloqueantes **MPI_Isend** pelas funções bloqueantes síncronas **MPI_Ssend**, precisamos fazer algumas modificações para garantir um código livre de *deadlock*. A mais importante dessas alterações é assegurar que, a cada passo do algoritmo, metade dos processos chame primeiro a rotina de envio de mensagens, enquanto a outra metade, destino dessas mensagens, chame primeiro a rotina de recepção de mensagens.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_ssend.c  */
int meu_ranque, num_procs, etiq=1, cont=1;
int pot2, destino, meu_valor;
int reducao, recebido;
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Aborta se o número de processos não for potência de 2 */
    pot2 = num_procs;
    while (((pot2 % 2) == 0) && pot2 > 1)
        pot2 /= 2;
    if (pot2 != 1) {
        if (meu_ranque == 0)
            printf("Por favor execute com número de processos potencia de 2 \n");
        MPI_Finalize();
        exit(0);
        }
    /* Cada processo tem um valor diferente para a redução */
    meu_valor = meu_ranque*num_procs;
    reducao = meu_valor;
    /* Calcula o destino no algoritmo de "recursive doubling" */
    for (int i = 1; i <= (num_procs/2); i += i) {
        if ((meu_ranque/i)%2 == 0) {
            destino = meu_ranque + i;
    /* Metade dos processos envia primeiro e recebe depois */
            MPI_Ssend(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD);
            MPI_Recv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &estado);
    /* Realiza operação de redução com os dados recebidos */
            if (recebido > reducao)
                reducao = recebido;
        } else {
            destino = meu_ranque-i;
    /* A outra metade dos processos recebe primeiro e envia depois */
            MPI_Recv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &estado);
            MPI_Ssend(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD);
    /* Realiza operação de redução com os dados recebidos */
            if (recebido > reducao)
                reducao = recebido;
```

```
        }
    }
    printf("Meu valor = %d, redução = %d \n", meu_valor, reducao);
    MPI_Finalize();
    return(0);
}
```

É preciso garantir que para cada **MPI_Ssend** haja sempre um **MPI_Recv** correspondente postado, para evitar a ocorrência de *deadlock*. Essa é uma forma de programação **segura**, podendo ser utilizada com qualquer combinação de tipos de rotinas de envio e recepção de mensagens do MPI. Com essas modificações, como era esperado, o programa obtém na saída os mesmos resultados que a execução do programa com funções não bloqueantes.

5.5.2 Modo pronto

No modo pronto, o envio só pode ser iniciado se houver uma rotina de recepção correspondente já postada. Caso isso não ocorra, o programa terminará com erro. Embora seja esperado, não é garantido que a implementação das rotinas em modo pronto seja mais eficiente que as de modo padrão. É necessário o uso de funções de sincronização (por exemplo barreiras) para garantir que a recepção em um processo seja postada antes do envio pelo outro. Esse é o modo mais difícil de programar e só deve ser usado quando o desempenho for importante. A sintaxe desta função é:

int MPI_Rsend(void* mensagem, int cont, MPI_Datatype tipo_mpi, int destino, int etiq, MPI_Comm com)

Se tomarmos da Seção 5.3 e substituirmos as funções não bloqueantes **MPI_Isend** pelas funções bloqueantes prontas **MPI_Rsend**, obteremos o programa ilustrado a seguir.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_rsend.c  */
int i, meu_ranque, num_procs;
int pot2, destino, meu_valor;
int reducao, recebido, etiq=1, cont=1;
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Request pedido;
    /* Aborta se o número de processos não for potência de 2 */
    pot2 = num_procs;
    while (((pot2 % 2) == 0) && pot2 > 1)
        pot2 /= 2;
    if (pot2 != 1) {
        if (meu_ranque == 0)
            printf("Por favor execute com número de processos potencia de 2 \n");
        MPI_Finalize();
        exit(0);
        }
    /* Cada processo tem um valor diferente para a redução */
    meu_valor = meu_ranque*num_procs;
    reducao = meu_valor;
    /* Calcula o destino no algoritmo de "recursive doubling" */
    for (i = 1; i <= (num_procs/2); i += i) {
        if ((meu_ranque/i)%2 == 0)
            destino = meu_ranque + i;
```

```
        else
            destino = meu_ranque - i;
    /* Todos os processos postam as recepções primeiro */
        MPI_Irecv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &pedido);
    /* A barreira assegura que todas as recepções já foram postadas */
        MPI_Barrier(MPI_COMM_WORLD);
    /* E agora todos os envios podem ser realizados */
        MPI_Rsend(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD);
    /* Fica em espera bloqueada aguardando os dados chegarem */
        MPI_Wait(&pedido, &estado);
    /* Realiza operação de redução com os dados recebidos */
        if (recebido > reducao)
            reducao = recebido;
    }
    printf("Meu valor = %d, redução = %d \n", meu_valor, reducao);
    MPI_Finalize();
    return(0);
}
```

Nesse exemplo todas as rotinas de recepção devem ser postadas previamente. A barreira garante que nenhum processo vai iniciar o envio das mensagens antes que todas as recepções correspondentes tenham sido postadas. Com esses cuidados, o resultado da execução do programa será o mesmo que com o uso de funções não bloqueantes.

5.5.3 Modo bufferizado

No modo "bufferizado" a operação de envio pode ser iniciada havendo ou não uma operação de recepção correspondente iniciada. A operação de envio pode terminar antes de uma recepção correspondente tenha sido postada. O uso do modo "bufferizado" permite que a comunicação seja assíncrona e garante que sempre haverá espaço suficiente para armazenar as mensagens, já que esse espaço é provido pela própria aplicação. A sua sintaxe é a seguinte:

int MPI_Bsend(void* mensagem, int cont, MPI_Datatype tipo_mpi, int dest, int etiq, MPI_Comm com)

As operações de envio e recepção no modo "bufferizado" garantidamente não são sincronizadas. É função do usuário, e não do sistema, gerenciar a alocação dos *buffers* com o uso de funções adicionais de bibliotecas da linguagem C (por exemplo, *malloc()*) e do MPI para alocação e liberação do espaço para armazenamento das mensagens. A função para informar ao MPI qual o *buffer* a ser utilizado é a seguinte:

int MPI_Buffer_attach(void *buffer, int tam_buffer)

Só pode haver um *buffer* ativo por vez. O total de espaço alocado deve ser suficiente para garantir o funcionamento correto do programa, ou seja, o valor passado no parâmetro *tam_buffer* deve ser maior que a soma do espaço necessário para todas as chamadas de **MPI_Bsend** que possam estar pendentes no programa, MAIS uma constante **MPI_BSEND_OVERHEAD** para cada **MPI_Bsend** pendente. Para efeito de cálculo do espaço necessário para cada envio, a rotina **MPI_Pack_size** deve ser usada.

int MPI_Pack_size(int cont, MPI_Datatype datatype, MPI_Comm com, int *tam)

Por exemplo, considere o seguinte código:

```
MPI_Buffer_attach(buffer, tam_buffer);
MPI_Bsend(mensagem1, 20, MPI_INT, destino1, etiq1, com);
MPI_Bsend(mensagem2, 40, MPI_FLOAT, destino2, etiq2,com);
```

O valor *tam_buffer* passado como parâmetro para a rotina **MPI_Buffer_attach** deve ser maior que o valor computado em:

```
MPI_Pack_size(20, MPI_INT, com, &tam1);
MPI_Pack_size(40, MPI_FLOAT, com, &tam2);
tam_buffer =  tam1 + tam2 + 2 * MPI_BSEND_OVERHEAD;
```

A constante **MPI_BSEND_OVERHEAD** especifica o máximo de espaço adicional possível de ser utilizado pela rotina **MPI_Bsend** para enviar cada mensagem. Essa constante tem um valor específico para cada implementação de MPI. Depois que o *buffer* não for mais necessário, isso deve ser informado ao MPI com a rotina:

int MPI_Buffer_detach (void *endereco_buffer, int *tam_buffer)

Note que essa rotina retorna dois parâmetros: um ponteiro para o *buffer* previamente alocado e um ponteiro para o seu tamanho, logo endereços devem ser passados como parâmetros. Esse espaço retornado pode eventualmente ser reutilizado ou desalocado. O espaço alocado para o *buffer* não é liberado automaticamente, devendo-se utilizar rotinas da biblioteca em C (por exemplo, *free()*) para isso.

Vemos no exemplo a seguir o uso das funções de envio bloqueantes bufferizadas **MPI_Bsend**, realizando uma operação de redução sobre um vetor de 4 posições.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define TAM 4
int main(int argc, char *argv[]) { /* mpi_bsend.c  */
int i, j, meu_ranque, num_procs;
int pot2, meu_valor, reducao[TAM];
int destino, recebido[TAM], etiq=1, cont=TAM;
MPI_Status estado;
void *buffer;
int tam_buffer;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Aborta se o número de processos não for potência de 2 */
    pot2 = num_procs;
    while (((pot2 % 2) == 0) && pot2 > 1)
           pot2 /= 2;
    if (pot2 != 1) {
        if (meu_ranque == 0)
            printf("Por favor execute com número de processos potencia de 2 \n");
        MPI_Finalize();
        exit(0);
    }
    /* Faz a alocação do espaço para o buffer de envio */
    MPI_Pack_size(TAM, MPI_INT, MPI_COMM_WORLD, &tam_buffer);
    tam_buffer = tam_buffer +  MPI_BSEND_OVERHEAD;
    buffer =(void*) malloc(tam_buffer);
    MPI_Buffer_attach(buffer, tam_buffer);
    if (meu_ranque == 0)
        printf("Alocados %d bytes \n", tam_buffer);
    /* Cada processo tem um valor diferente para a redução */
```

```
    meu_valor = meu_ranque*num_procs;
    for (j = 0; j < TAM; j++)
         reducao[j]  = meu_valor;
    /* Calcula o destino no algoritmo de "recursive doubling" */
    for (i = 1; i <= (num_procs/2); i += i) {
         if ((meu_ranque/i)%2 == 0)
             destino = meu_ranque + i;
         else
             destino = meu_ranque - i;
    /* Faz as operações de envio e recepção, nesta ordem */
    MPI_Bsend(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD);
    MPI_Recv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &estado);
   /* Realiza operação de redução com os dados recebidos */
    for (j = 0; j < TAM; j++)
         if (recebido[j] > reducao[j])
             reducao[j] = recebido[j];
    }
    printf("Meus valores = %d %d %d %d \t", meu_valor, meu_valor, meu_valor, meu_valor);
    printf("Redução = ");
    for (j = 0; j < TAM; j++)
         printf(" %d", reducao[j]);
    printf("\n");
    /* Note o uso do & na passagem de parâmetros */
    MPI_Buffer_detach(&buffer, &tam_buffer);
    free (buffer);
    MPI_Finalize();
    return(0);
}
```

Note que todos processos podem enviar a sua mensagem e depois esperar pela recepção. A garantia de que esse programa funcionará corretamente se deve a existência dos *buffers* alocados explicitamente pela rotina **MPI_Buffer_attach**. Um possível resultado para a execução com 8 processos, seria o apresentado a seguir. Observe que os dados propriamente ditos só requerem 16 bytes, sendo os 128 bytes adicionais por conta do envelope da mensagem (ranque de origem, etiqueta, comunicador, tamanho e tipo dos dados etc.) expressos pela constante **MPI_BSEND_OVERHEAD**.

```
Alocados 144 bytes
Meus valores = 48 48 48 48      Redução =  56 56 56 56
Meus valores = 56 56 56 56      Redução =  56 56 56 56
Meus valores = 0 0 0 0          Redução =  56 56 56 56
Meus valores = 8 8 8 8          Redução =  56 56 56 56
Meus valores = 16 16 16 16      Redução =  56 56 56 56
Meus valores = 24 24 24 24      Redução =  56 56 56 56
Meus valores = 32 32 32 32      Redução =  56 56 56 56
Meus valores = 40 40 40 40      Redução =  56 56 56 56
```

5.5.4 Modo padrão

Nesse modo, as operações de envio e recepção são bloqueantes, porém a implementação do MPI decide, em função do tamanho da mensagens, se elas serão enviadas com uso de *buffers* ou transmitidas no modo síncrono. Uma rotina de envio no modo padrão pode ser iniciada havendo ou não uma rotina de recepção correspondente postada. Porém, não se pode assumir que a operação de envio terminará antes ou depois de a recepção correspondente ter sido iniciada. Como consequência, é possível que o programa funcione bem em um sistema, mas não em outro, caso o programa esteja programado de uma forma **não segura**.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_padrao.c  */
int i, meu_ranque, num_procs;
int pot2, destino, meu_valor;
int reducao, recebido, etiq=1, cont=1;
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Aborta se número de processos não for potência de 2*/
    pot2 = num_procs;
    while (((pot2 % 2) == 0) && pot2 > 1)
           pot2 /= 2;
    if (pot2 != 1) {
        if (meu_ranque == 0)
            printf("Por favor execute com número de processos potencia de 2 \n");
        MPI_Finalize();
        exit(0);
        }
    /* Cada processo tem um valor diferente */
    meu_valor = meu_ranque*num_procs;
    reducao = meu_valor;
    /* Calcula o destino no algoritmo de "recursive doubling" */
    for (i = 1; i <= (num_procs/2); i += i) {
        if ((meu_ranque/i)%2 == 0)
            destino = meu_ranque + i;
        else
            destino = meu_ranque-i;
    /* Realiza as operações de envio primeiro. Vai na fé! */
        MPI_Send(&reducao, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD);
    /* Realiza as operações de recepção. Se tudo deu certo, consegue chegar aqui. */
        MPI_Recv(&recebido, cont, MPI_INT, destino, etiq, MPI_COMM_WORLD, &estado);
    /* Realiza a operação de redução */
        if (recebido > reducao)
            reducao = recebido;
    }
    printf("Meu valor = %d, redução = %d \n", meu_valor, reducao);
    MPI_Finalize();
    return(0);
}
```

No exemplo apresentado, a forma de programação utilizada é dita **não segura** (veja a Seção 5.6). Isto é, pode funcionar em alguns casos e em outros não, dependendo de o sistema utilizar ou não *buffers* para o envio das mensagens. A ordenação correta das rotinas de envio e recepção, como no exemplo com uso das funções síncronas, garante o funcionamento do programa em qualquer situação.

5.6 Evitando o impasse ou deadlock

No Capítulo 2 apresentamos o problema da ocorrência de *deadlock* na programação paralela e as formas de evitá-lo em um ambiente de memória compartilhada (Herlihy and Shavit 2012). No modelo de programação paralela por troca de mensagem esse problema também pode ocorrer. Isso se deve basicamente pela ocorrência de ciclos de dependência entre os processos durante a troca de mensagens entre eles.

Por exemplo, ao enviarmos uma mensagem grande do ***processo 0*** para o ***processo 1***, verificamos que o espaço de armazenamento no destino é insuficiente. Nesse caso a rotina de envio deve esperar até que o usuário providencie espaço de memória suficiente (através da chamada de uma rotina de recepção). O que acontece com este código?

Tabela 5.2: Evitando *deadlock* - possível *deadlock*

Processo 0	**Processo 1**
MPI_Send(1)	MPI_Send(0)
MPI_Recv(1)	MPI_Recv(0)

Esse modo de programação é chamado de **inseguro** porque depende da disponibilidade dos *buffers* de sistema, sem os quais é certa a ocorrência de *deadlock*. Algumas soluções podem ser utilizadas para evitarmos que isso ocorra, tais como:

1. Ordenar as operações de envio adequadamente (Tabela 5.3);

Tabela 5.3: Ordenação adequada das rotinas de envio e recepção

Processo 0	**Processo 1**
MPI_Send(1)	MPI_Recv(0)
MPI_Recv(1)	MPI_Send(0)

2. Fornecer explicitamente um *buffer* para envio da mensagem (Tabela 5.4);

Tabela 5.4: Uso de rotinas bufferizadas

Processo 0	**Processo 1**
MPI_Bsend(1)	MPI_Bsend(0)
MPI_Recv(1)	MPI_Recv(0)

3. Utilizar operações de envio e recepção não bloqueantes (Tabela 5.5);

Tabela 5.5: Uso de rotinas não bloqueantes

Processo 0	**Processo 1**
MPI_Isend(1)	MPI_Isend(0)
MPI_Irecv(1)	MPI_Irecv(0)
MPI_Waitall	MPI_Waitall

4. Fornecer um *buffer* de recepção ao mesmo tempo que se envia a mensagem com o uso da função **MPI_Sendrecv**, como visto na Tabela 5.6.

Tabela 5.6: Uso de MPI_Sendrecv

Processo 0	**Processo 1**
MPI_Sendrecv(1)	MPI_Sendrecv(0)

Essa última função permite envio e recepção simultâneos, com tipos de dados de envio e recepção diferentes. Podemos usar **MPI_Sendrecv** com um **MPI_Recv** ou **MPI_Send** comuns (ou **MPI_Irecv**, **MPI_Ssend** etc.). Veja a seguir a sintaxe dessa rotina:

int MPI_Sendrecv(const void *vet_envia, int cont_envia, MPI_Datatype tipo_envia, int dest, int etiq_envia, void *vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, int origem, int etiq_recebe, MPI_Comm com, MPI_Status *estado)

Os parâmetros têm o mesmo significado e uso que nas respectivas rotinas de envio e recepção. Podemos pensar nessa rotina como sendo um **MPI_Isend**, seguido de um **MPI_Irecv**, e um par de **MPI_Waits**. Assim, efetivamente, o envio e a recepção da mensagem acontecem em paralelo. O exemplo a seguir apresenta um programa bem simples que faz uso dessas rotinas, onde os ***processos 0 e 1*** trocam um valor inteiro. Esse exemplo deve ser executado com apenas 2 processos.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) { /* mpi_sendrecv.c  */
int meu_ranque, num_procs, destino, etiq=0, raiz=0;
int dados, dados_locais;
MPI_Status estado;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Se o número de processos não for igual a 2 então aborta */
    if (num_procs != 2) {
        if (meu_ranque == raiz)
            printf("Por favor execute com apenas 2 processos  \n");
        MPI_Finalize();
        exit(0);
    }
    /* Valor a ser enviado */
    dados = meu_ranque*2 + 1;
    printf("Processo %d: tem dado %d \n", meu_ranque, dados);
    /* Determina o destino */
    destino = (meu_ranque == raiz)? 1 : 0;
      MPI_Sendrecv(&dados, 1, MPI_INT, destino, etiq, &dados_locais, 1, MPI_INT, destino,
etiq, MPI_COMM_WORLD, &estado);
    printf("Processo %d:  recebeu o  dado: %d\n", meu_ranque, dados_locais);
    MPI_Finalize();
    return(0);
}
```

O resultado da execução desse programa com 2 processos é o seguinte:

```
Processo 0: tem dado 1
Processo 1: tem dado 3
Processo 0:  recebeu o  dado: 3
Processo 1:  recebeu o  dado: 1
```

5.7 Considerações de desempenho

O MPI tem diversos modos de envio de mensagens. Esses modos representam diferentes escolhas de "bufferização" (onde os dados são armazenados até a sua recepção) e sincronização (quando um envio completa).

O termo não bloqueante se refere apenas à possibilidade de os dados do *buffer* da aplicação serem reutilizados depois da chamada. Em nenhum momento a especificação do MPI determina que as operações de comunicação devam ocorrer em paralelo às de computação.

É possível obtermos um melhor desempenho se os programas forem escritos usando as rotinas **MPI_Ssend**. Isso ocorre porque são evitadas movimentações adicionais de dados, como no caso do uso da "bufferização", que claramente impõem uma penalidade maior para o tempo de envio e recepção das mensagens. Se for utilizado o **MPI_Send**, a decisão sobre "bufferizar" (ou não) os dados é deixada para a implementação do MPI, podendo resultar em um programa que não funcione corretamente em algumas situações, por exemplo, quando o tamanho das mensagens enviadas é muito grande.

Quando utilizamos a comunicação não bloqueante, a rotina devolve um objeto do tipo *MPI_Request*, utilizado para se verificar posteriormente se a comunicação terminou ou não. É importante sempre verificarmos se a comunicação terminou normalmente, não apenas para reutilizar os *buffers* da mensagem, mas para liberar os recursos que o MPI aloca para verificar o *status* da comunicações não bloqueantes, que são mantidos enquanto não for notificado para a aplicação do usuário que a comunicação completou normalmente.

Quando utilizar as rotinas de tempo **MPI_Wtime()** lembre-se de que o resultado pode ser afetado por todas as rotinas de envio de mensagens pendentes. Desse modo, dados que ainda não tenham sido recebidos pelas rotinas correspondentes, sejam elas não bloqueantes ou do tipo padrão, podem afetar essas medições. Via de regra execute os seguintes passos:

- Receba e complete todas as transferências em um comunicador, lembrando que as funções coletivas fazem isso automaticamente;
- Em seguida, chame a função **MPI_Barrier** naquele comunicador;
- Chame a rotina **MPI_Wtime** em alguns ou todos os processos. Todas as chamadas devem medir o mesmo tempo decorrido.

A seguir destacamos as características dos diversos modos das rotinas de envio do MPI:

- **MPI_Send**: não retorna até que os dados da aplicação possam ser reutilizados. Pode ser "bufferizado" ou não pelo sistema, dependendo do tamanho da mensagem.
- **MPI_Bsend**: usa um *buffer* adicional definido pelo usuário, retorna imediatamente após a cópia dos dados, que podem ser reutilizados imediatamente.
- **MPI_Ssend**: não retorna até que uma rotina de recepção correspondente tenha sido postada.
- **MPI_Rsend**: pode ser usada APENAS se a rotina de recepção correspondente foi postada previamente. É responsabilidade do usuário escrever um programa correto.
- **MPI_Isend**: faz o envio não bloqueante, mas não necessariamente assíncrono. Os dados da aplicação não podem ser reutilizados até que uma rotina **MPI_Wait ou MPI_Test** tenha retornado um manipulador do tipo MPI_Request.
- **MPI_Ibsend**: faz o envio "bufferizado" não bloqueante.
- **MPI_Issend**: faz o envio síncrono e não bloqueante.
- **MPI_Irsend**: faz o mesmo que **MPI_Rsend**, mas não é bloqueante.

Procure usar **MPI_Isend** ou **MPI_Irecv** caso o uso de rotinas não bloqueantes seja necessário. Nesse caso, existe a possibilidade de sobreposição das operações de comunicação com computação e deve ser uma opção a ser considerada quando isso trouxer algum benefício.

Recomenda-se o uso de **MPI_Bsend** apenas quando não for possível usar o **MPI_Isend**. As demais rotinas, **MPI_Rsend**, **MPI_Issend** etc. são usadas muito raramente. No caso de uso da rotina **MPI_Issend**, uma função **MPI_Wait** ou **MPI_Test** com o manipulador *pedido* só completará quando uma rotina de recepção correspondente for postada.

Finalmente, apresentamos a seguir um resumo das características, vantagens e desvantagens de cada modo de comunicação das rotinas de envio e recepção do MPI.

- **Modo Síncrono**
  - Vantagens: Mais seguro e, portanto, mais portátil. A ordem *Envio/Recepção* não é crítica. O total de espaço gasto é irrelevante.
  - Desvantagens: Pode haver substancial sobrecarga de sincronização, levando a uma perda de desempenho.
- **Modo Pronto**
  - Vantagens: A sobrecarga total é a mais baixa. O protocolo de quatro fases para *Envio/Recepção* não é necessário.
  - Desvantagens: A rotina de recepção deve preceder a de envio.
- **Modo Bufferizado**
  - Vantagens: Desacopla o envio da recepção. O programador pode controlar o tamanho do *buffer*. A ordem de *Envio/Recepção* é irrelevante.
  - Desvantagens: Existe a sobrecarga adicional do sistema para copiar o *buffer*.
- **Modo Padrão**
  - Vantagens: É adequado na maioria dos casos.
  - Desvantagens: Seu programa pode não funcionar corretamente se as mensagens forem muito grandes.

5.8 Estudo de caso: números primos

Nesta seção, nosso estudo de caso será um programa para calcular a quantidade de números primos entre 0 e um determinado valor inteiro Math input error. É um programa bem simples e apresentamos no próximo exemplo o código fonte da versão sequencial do programa. Ele basicamente verifica se Math input error é divisível por algum número ímpar entre 0 e a raiz quadrada de Math input error, sendo que os números pares são descartados de imediato.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int primo (long int n) {
```

```
long int i;

    for (i = 3; i < (int)(sqrt(n) + 1); i+=2)
         if (n%i == 0)
             return 0;
    return 1;
}
int main(int argc, char *argv[]) { /* primos_seq.c  */
int total=0;
long int i, n;

    if (argc < 2) {
        printf("Valor inválido! Entre com o valor do maior inteiro\n");
        return 0;
    } else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    for (i = 3; i <= n; i += 2)
         if(primo(i) == 1) total++;
    /* Acrescenta o dois, que também é primo */
    total += 1;
    printf("Quant. de primos entre 1 e n: %d \n", total);
    return(0);
}
```

Como na paralelização do problema do cálculo da integral pelo método do trapézio, as tarefas poderiam ser distribuídas em lote, passando-se uma faixa contínua de valores para serem verificados por cada processo. Mas descartamos de início esse tipo de distribuição, pois a busca dos primos para valores de Math input error maiores leva mais tempo, ocasionando um grande desbalanceamento de carga entre os processos.

Uma alternativa possível para resolver esse problema é fazer com que cada processo comece verificando um número ímpar diferente, saltando (Math input error) números entre cada número ímpar verificado, alternadamente. Esse método de passar os números ímpares alternadamente diminui as chances de ocorrer um desbalanceamento de carga entre os processos, já que todos os processos vão avaliar tanto números pequenos, como os de grande valor.

Esse programa é classificado como embaraçosamente paralelo porque possui apenas uma chamada de comunicação coletiva usada para reduzir o número de primos encontrados. Vejamos a seguir o código-fonte para esta versão do programa.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <math.h>
int primo (long int n) {
    long int i;

    for (i = 3; i < (long int)(sqrt(n) + 1); i+=2) {
            if(n%i == 0) return 0;
    }
    return 1;
}
int main(int argc, char *argv[]) { /* mpi_primos.c  */
    double t_inicial, t_final;
    int cont = 0, total = 0;
    long int i, n;
    int meu_ranque, num_procs, inicio, salto;
```

```
    if (argc < 2) {
        printf("Valor inválido! Entre com um valor do maior  inteiro\n");
        return 0;
    } else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    t_inicial = MPI_Wtime();
    inicio = 3 + meu_ranque*2;
    salto = num_procs*2;
    for (i = inicio; i <= n; i += salto) {
         if(primo(i) == 1) cont++;
    }
    if (num_procs > 1) {
        MPI_Reduce(&cont, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    } else {
        total = cont;
    }
    if (meu_ranque == 0) {
        t_final = MPI_Wtime(); /* Só o processo 0 mede o tempo */
        /* Acrescenta o dois, que também é primo */
        total += 1;
        printf("Quant. de primos entre 1 e n: %d \n", total);
        printf("Tempo de execucao: %1.3f \n", t_final - t_inicial);
    }
    MPI_Finalize();
    return(0);
}
```

Esse programa deve ser salvo com o nome *mpi_primos.c*. Em seguida deve ser compilado (não esquecer a opção *-lm*) e executado com os seguintes comandos em um terminal de um sistema operacional do tipo Linux:

```
$ mpicc -O3 -o primos mpi_primos.c -lm
$ mpirun -np 4 ./primos 10000000
```

O resultado da execução desse programa será o seguinte:

```
Quant. de primos entre 1 e n: 664579
Tempo de execucao: 0.783
```

Avaliação de desempenho

Uma arquitetura de multiprocessamento simétrico com um total de 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz) foi escolhida para realizar o nosso estudo de escalabilidade.

O gráfico da Figura 5.3 apresenta os tempos de execução para esta versão paralela do programa, com o número de processos variando entre 2 e 24, para valores de Math input error iguais a 10.000.000 e 100.000.000. Contudo, podemos observar neste gráfico uma grande irregularidade nos tempos de execução, dependendo do número de processos utilizados.

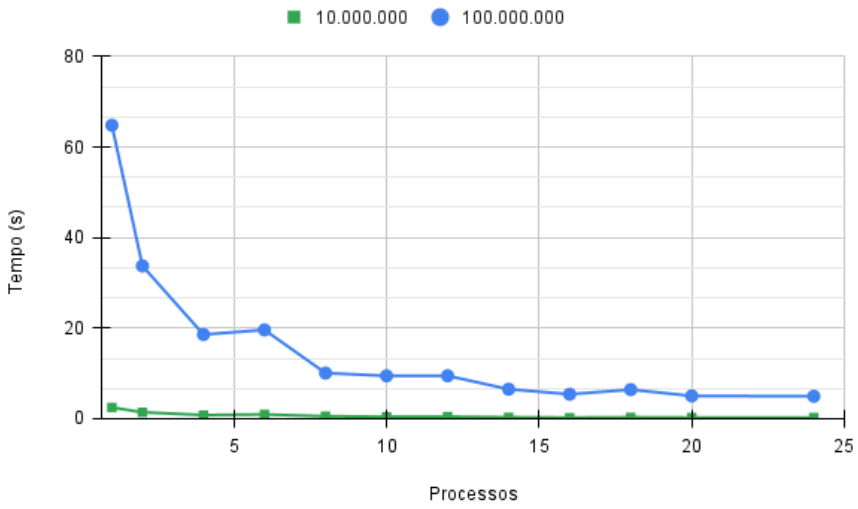

Figura 5.3: Tempo de execução - Primos

Uma análise mais detalhada nos gráficos de *speedup* (Figura 5.4) e eficiência (Figura 5.5) comprova uma perda de aceleração e eficiência quando o número de processos não é uma potência exata de 2. Ou seja, apesar dos cuidados tomados, o comportamento do programa ainda não está adequado, pois há um evidente desbalanceamento de carga nesses casos.

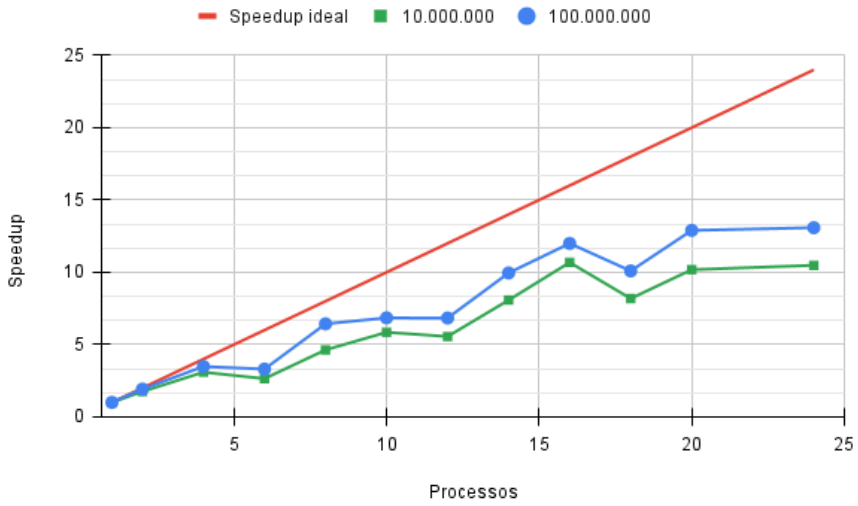

Figura 5.4: Speedup - Primos

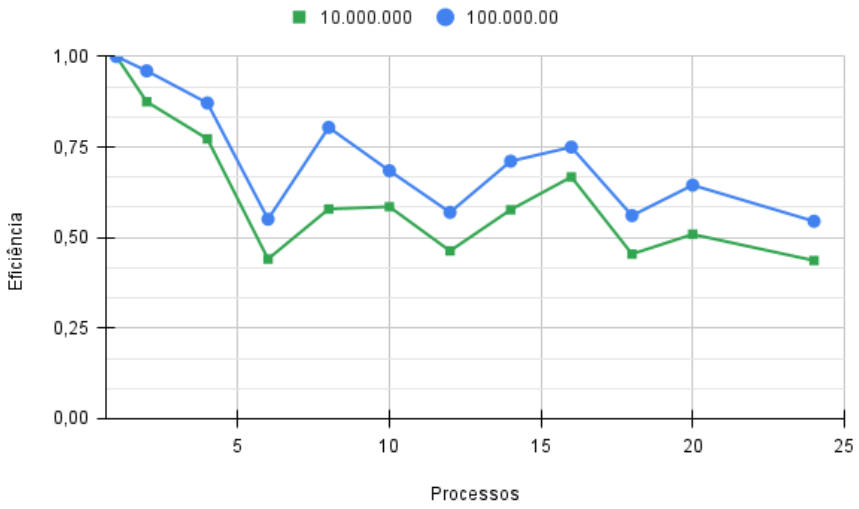

Figura 5.5: Eficiência - Primos

Curiosamente, em nossa análise, descobrimos que em alguns casos, certos processos verificam em toda a sua execução apenas números que são múltiplos de números primos, ou seja, não primos, fazendo com que esses processos terminem a sua execução muito mais rapidamente que os demais. Logo, o desbalanceamento de carga que queríamos evitar reaparece nesses casos.

Como podemos resolver esse problema? Como se trata de um programa do tipo embaraçosamente paralelo, claramente não se trata de um problema de comunicação, ou de perdas por sincronização. Como observarmos, há uma irregularidade muito grande no gráfico de eficiência para valores do número de processadores (*num_procs*) iguais a 3, 5, 6, 7, 9, 10, 11, 12, 13 e 15; quando alguns processos terminam muito antes dos demais, que ficam sobrecarregados.

Para vencer essa dificuldade empregamos um método de balanceamento dinâmico chamado de "saco de tarefas". Nesse método, um processo (mestre) fica responsável por enviar as tarefas para os demais processos (trabalhadores). Assim que uma tarefa é terminada e o resultado é enviado para o mestre, uma outra tarefa será alocada para o trabalhador, e assim sucessivamente, até que não haja mais tarefas para serem executadas.

Como forma de evitar uma sobrecarga na comunicação, esse método requer que cada tarefa tenha um tamanho mínimo, para manter o processador ocupado por um tempo razoavelmente grande. Vejamos a seguir uma proposta de código que aplica esse método.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#define TAMANHO 500000
int primo (int n) {
int i;
    for (i = 3; i < (int)(sqrt(n) + 1); i+=2) {
         if (n%i == 0) return 0;
    }
    return 1;
}
int main(int argc, char *argv[]) { /* mpi_primosbag.c  */
double t_inicial, t_final;
int i, n, cont = 0, total = 0;
```

```
int meu_ranque, num_procs, inicio, dest, raiz=0, tag=1, stop=0;
MPI_Status estados;
/* Verifica o número de argumentos passados */
    if (argc < 2) {
        printf("Entre com o valor do maior inteiro como parâmetro para o programa.\n");
        return 0;
    } else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
/* Se houver menos que dois processos aborta */
    if (num_procs < 2) {
        printf("Este programa deve ser executado com no mínimo dois processos.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
        return(1);
    }
/* Registra o tempo inicial de execução do programa */
    t_inicial = MPI_Wtime();
/* Envia pedaços com TAMANHO números para cada processo */
    if (meu_ranque == 0) {
        for (dest=1, inicio=3; dest < num_procs && inicio < n; dest++, inicio += TAMANHO)
             MPI_Send(&inicio, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
/* Fica recebendo as contagens parciais de cada processo */
        while (stop < (num_procs-1)) {
                 MPI_Recv(&cont, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&estado);
             total += cont;
             dest = estado.MPI_SOURCE;
             if (inicio > n) {
                 tag = 99;
                 stop++;
             }
/* Envia um novo pedaço com TAMANHO números para o mesmo processo*/
             MPI_Send(&inicio, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
             inicio += TAMANHO;
        }
    } else {
/* Cada processo escravo recebe o início do espaço de busca */
        while (estado.MPI_TAG != 99) {
            MPI_Recv(&inicio, 1, MPI_INT, raiz, MPI_ANY_TAG, MPI_COMM_WORLD, &estado);
            if (estado.MPI_TAG != 99) {
                for (i = inicio, cont=0; i < (inicio + TAMANHO) && i < n; i+=2)
                     if (primo(i) == 1)
                         cont++;
/* Envia a contagem parcial para o processo mestre */
                MPI_Send(&cont, 1, MPI_INT, raiz, tag, MPI_COMM_WORLD);
            }
        }
/* Registra o tempo final de execução */
    t_final = MPI_Wtime();
    }
    if (meu_ranque == 0) {
        t_final = MPI_Wtime();
        total += 1;     /* Acrescenta o 2, que é primo */
        printf("Quant. de primos entre 1 e %d: %d \n", n, total);
        printf("Tempo de execucao: %1.3f \n", t_final - t_inicial);
    }
/* Finaliza o programa */
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
```

```
    return(0);
}
```

Os tempos de execução desta versão podem ser vistos na Tabela 5.7 e na Figura 5.6. Nessa versão, a quantidade de mensagens enviadas é maior, mas o tempo total de execução não aumenta e o comportamento do programa é melhor como um todo, apresentando um gráfico com curvas mais bem comportadas, independentemente do número de processos utilizados. O tempo de 1055 segundos com 1 *thread* e Math input error foi utilizado como base para os demais cálculos da tabela. Um valor alto para Math input error no experimento foi adotado para haja uma carga proporcionalmente alta visando uma melhor análise do comportamento da solução.

Tabela 5.7: Métricas números primos - Saco de Tarefas

Número de processos	**Tempo de execução**	***Speedup***	**Eficiência**
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

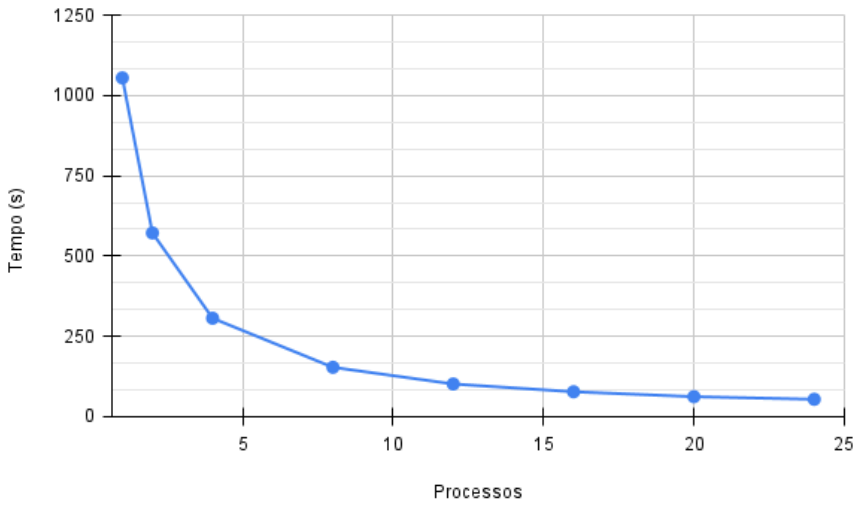

Figura 5.6: Tempo de Execução - Saco de Tarefas

As rotinas de envio e recepção utilizadas são do modo padrão, ou seja, para mensagens de pequeno tamanho o envio e recepção são *bufferizados*, permitindo que o programa funcione corretamente, sem riscos de ocorrência de *deadlock*. A quantidade de números que é enviada para cada processo trabalhador verificar por vez, ou seja, o tamanho do lote, é determinado por tentativa e erro. Como dito, não pode ser muito pequeno, porque um grande número de mensagens seria gerado para uma pequena carga de trabalho; e também não pode ser muito grande, pois assim voltaríamos à condição original de desbalanceamento de carga.

A avaliação de desempenho que realizamos considera somente o número de processos trabalhadores, uma vez que o processo mestre não participa diretamente da verificação da quantidade de números primos.

Uma análise mais detalhada nos gráficos de *speedup* (Figura 5.7) e eficiência (Figura 5.8) comprova uma distribuição de carga mais equilibrada, principalmente para um valor de *N* igual a 700.000.000, onde a carga de trabalho atribuída para cada processo compensa os custos com comunicação e sincronização.

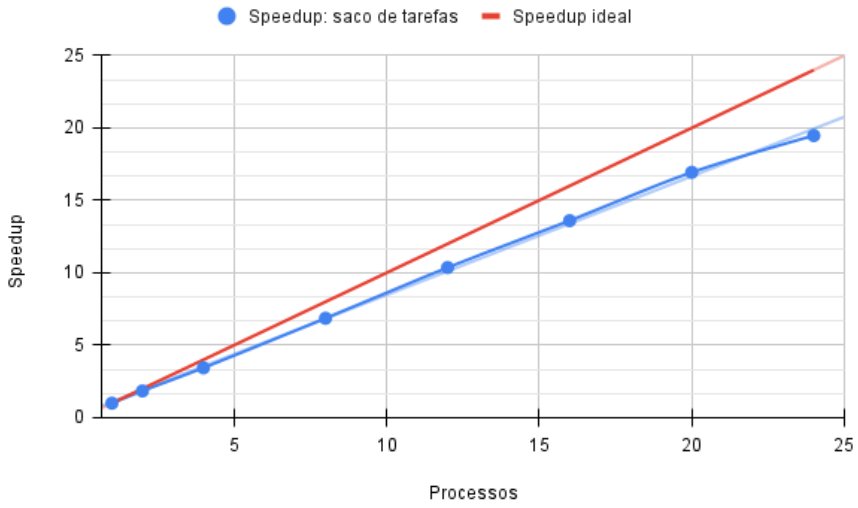

Figura 5.7: Speedup - Saco de Tarefas

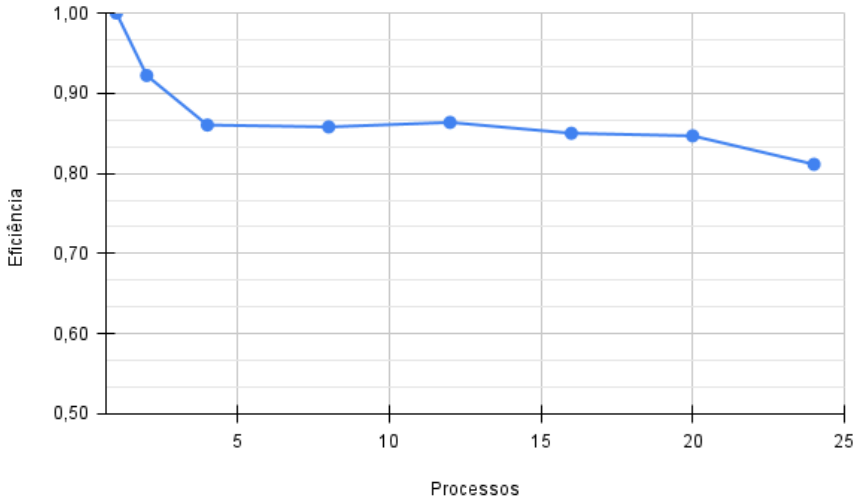

Figura 5.8: Eficiência - Saco de Tarefas

Tabela 5.8: Métricas números primos - saco de tarefas Gustafson

Tamanho N	**Tempo Seq.**	**Processos (*p*)**	**Tempo Paralelo**	**Speedup**	**Efici- ência**
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Tamanho N	Tempo Seq.	Processos (*p*)	Tempo Paralelo	Speedup	Efici- ência
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Desenhamos o gráfico da escalabilidade da solução balanceada na Figura 5.9. Essa linha se manteve semelhante à anterior (da Figura 5.7) e bem próximo da linha de *speedup* ideal.

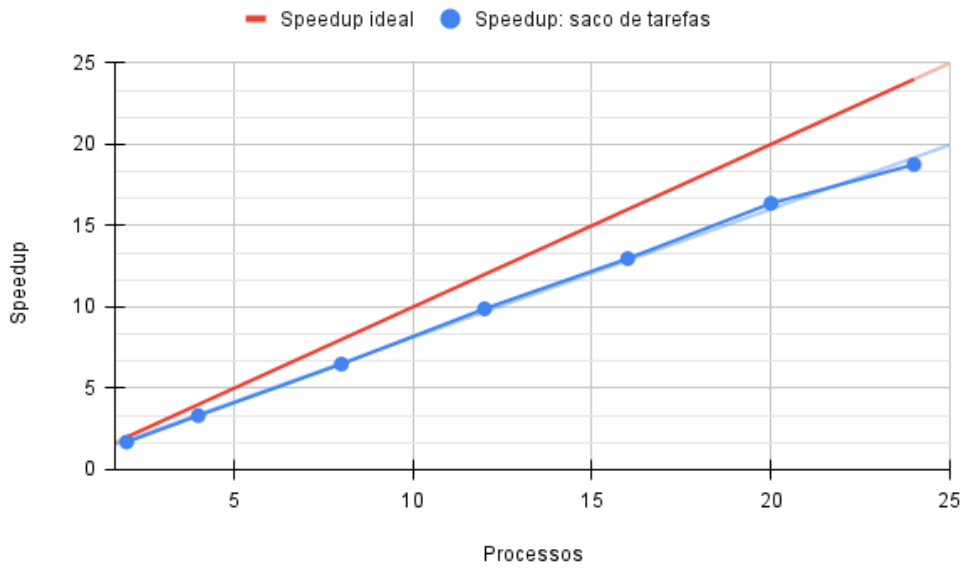

Figura 5.9: Gráfico de escalabilidade - números primos

O mesmo comportamento se mantém no gráfico de eficiência, conforme mostra a Figura 5.10. Há uma natural queda da eficiência a partir de 2 processos, resultado do impacto da parte sequencial sobre a solução (especialmente devido à comunicação entre os processos), mantendo a eficiência por volta de 80%.

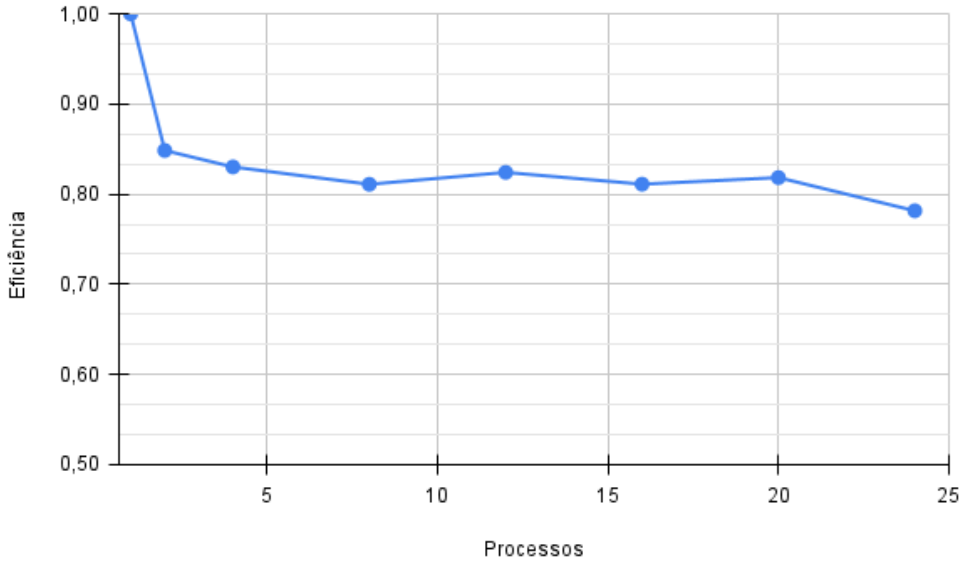

Figura 5.10: Gráfico de comparação de eficiência - números primos

5.9 Exercícios propostos

1. Com relação ao envio e recepção de mensagens no MPI, responda:
    1. O que acontece no MPI quando as mensagens de envio e recepção não podem ser perfeitamente sincronizadas?
    2. Quais as garantias que o programador tem quando utiliza rotinas de envio e recepção bloqueantes no MPI?
    3. Quais as características e vantagens do uso de rotinas de comunicação não bloqueantes no MPI?
    4. Qual a função e quais as diferenças entre as rotinas **MPI_Wait** e **MPI_Test**?
    5. Quais os requisitos de "bufferização" para uso das rotinas de envio no modo síncrono?
2. Reescreva o exemplo da Seção 5.4, com a função **MPI_Wait**, utilizando a rotina **MPI_Test**.
3. Reescreva o exemplo da Seção 5.4, com a função **MPI_Wait**, utilizando a rotina **MPI_Waitall**.
4. Com relação aos modos das rotinas de envio e recepção de mensagens no MPI, responda:
    1. É possível afirmar que um programa está seguro se esse executar corretamente utilizando apenas rotinas de envio no modo síncrono?
    2. Quais as restrições para a utilização das rotinas de envio no modo pronto? O seu uso é mais eficiente do que as rotinas no modo padrão?

3. Como é feito o cálculo do espaço necessário para o *buffer* de envio e recepção das mensagens no modo "bufferizado"?

4. Qual o critério principal para a utilização ou não de *buffers* no envio de mensagens no modo padrão?

5. Relacione quais as vantagens e desvantagens do uso das rotinas no MPI nos seguintes modos:

    1. No modo síncrono do MPI.

    2. No modo pronto do MPI.

    3. No modo "bufferizado" do MPI.

    4. No modo padrão do MPI.

6. Quais as soluções que podem ser tomadas para evitar a ocorrência de *deadlock* quando do envio e recepção de mensagens no MPI?

7. Reescreva o trecho de programa a seguir, de forma **segura**, utilizando rotinas de envio:

    1. não bloqueantes

    2. "bufferizadas"

    3. síncronas

```
#define TAM_MAX 65536

     sucessor = (meu_ranque + 1) % num_procs;
     predecessor = (meu_ranque - 1 + num_procs) % num_procs;
     MPI_Send(&a, 1, MPI_FLOAT, sucessor, 1, MPI_COMM_WORLD);
     MPI_Send(vetor, TAM_MAX, MPI_FLOAT, sucessor, 1, MPI_COMM_WORLD);
     MPI_Recv(&a, 1, MPI_FLOAT, predecessor, 1, MPI_COMM_WORLD);
     MPI_Recv(vetor, TAM_MAX, MPI_FLOAT, predecessor, 1, MPI_COMM_WORLD);
```

8. Reescreva o código da Seção 5.3 com uso da função MPI_Sendrecv.

9. Execute o programa da Seção 5.5, que faz uso da função **MPI_Bsend**, com diferentes números de processos (1, 2, 4 e 8 por exemplo) e verifique quais os valores de *tam_buffer* em cada um dos casos. Qual valor deve ser usado para **MPI_BSEND_OVERHEAD** em cada caso?

10. Modifique o código da Seção 5.5, que faz uso da função **MPI_Bsend**, para uso do modo padrão, aumentando o tamanho do *buffer* (contante TAM) e verifique qual o tamanho máximo com que o programa ainda consegue funcionar.

11. Modifique o código da Seção 5.5, que faz uso da função **MPI_Rsend**, para uso das rotinas no modo padrão e um tamanho de *buffer* maior que o verificado no exercício anterior e observe se o programa funciona corretamente ou não.

12. Qual o problema com o trecho de código a seguir?

```
int main(int argc, char *argv[]) {
float a[TAM][TAM] =
    {1.0, 2.0, 3.0, 4.0,
```

```
    5.0, 6.0, 7.0, 8.0,
    9.0, 10.0, 11.0, 12.0,
    13.0, 14.0, 15.0, 16.0};
float b[TAM];
int i, origem = 0;

     if (num_procs == TAM) {
         if (meu_ranque == 0) {
             for (i = 0; i < num_procs; i++)
                  MPI_Send(&a[i][0], 4, MPI_FLOAT, i, etiq, MPI_COMM_WORLD);
         }
         MPI_Recv(b, TAM, MPI_FLOAT, origem, etiq, MPI_COMM_WORLD, &estado);
                  printf("Ranque=  %d  b=  %3.1f  %3.1f  %3.1f  %3.1f\n",  meu_ranque,
b[0],b[1],b[2],b[3]);
     }
     else
         printf("Deve-se especificar %d processos. Abortando.\n", TAM);
     return(0);
}
```

13. A Conjectura de Collatz foi desenvolvida pelo matemático alemão Lothar Collatz. Nela, escolhendo-se um número natural inicial Math input error, onde Math input error, os seguintes critérios serão obedecidos: Se Math input error for par o seu sucessor será a metade e se Math input error for ímpar o seu sucessor será o triplo mais um, gerando então um novo número. Esse processo repete-se até que eventualmente se atinja o número 1.

    Use o modelo "saco de tarefas" para resolver esse problema em paralelo e verificar se os números inteiros no intervalo de 1 a 100.000.000 atendem a esses critérios e o maior número de passos necessários para chegar até o número 1. Atribua a um dos processos o papel de "mestre" (gerencia a distribuição de tarefas) e aos demais processos o papel de "trabalhadores" (executam a tarefa de avaliar se um número está dentro da conjectura). Utilize mensagens de envio e recepção não bloqueantes de forma a paralelizar a computação com o envio de mensagens.

6 OpenMP

Neste capítulo vamos apresentar o OpenMP e o seu conjunto de diretivas, cláusulas e funções mais utilizadas. Apresentaremos também o conceito de regiões paralelas, extração de paralelismo em laços e uso das diretivas e funções de sincronização do OpenMP.

6.1 Introdução

O OpenMP faz uso do paradigma de programação de memória compartilhada, no qual várias *threads* executam um programa em paralelo fazendo a comunicação através de variáveis compartilhadas em memória. O OpenMP e o MPI utilizam o mesmo algoritmo de decomposição do domínio de dados, mas a implementação do OpenMP é mais simples, porque não tem necessidade de troca de mensagens, e os dados globais podem ser lidos e escritos por qualquer *thread*.

No OpenMP é necessário paralelizar apenas as partes do código computacionalmente mais intensivas, não há necessidade de se converter o código inteiro. A maior fonte de extração de paralelismo são os laços, onde é realizada a maior parte do processamento. Desse modo, os trechos de pré-processamento e pós-processamento do programa, com baixa carga computacional, podem ser deixados à parte.

O OpenMP faz uso de um conjunto de diretivas passadas para o compilador, assim como de algumas funções, para explorar o paralelismo no código em linguagem C, C++ ou FORTRAN com o uso de *threads* (Hermanns 2002). Essas *threads* trabalham cooperativamente para a resolução de um problema explorando o paralelismo, normalmente através da adição de diretivas a um código sequencial convencional.

Uma diretiva é uma linha especial de código-fonte com significado especial apenas para determinados compiladores. Uma diretiva se distingue pela existência de uma sentinela no começo da linha. As sentinelas do OpenMP são:

C/C++:
#pragma omp
Fortran:
!$OMP (ou C$OMP ou *$OMP)

As diretivas podem ser acompanhadas de cláusulas declaradas logo após na mesma linha, com informação adicional para a diretiva do OpenMP. Nos exemplos apresentados neste livro utilizaremos apenas os *pragmas* e exemplos relativos à linguagem de programação C. Adicionalmente, para uso das funções do OpenMP que serão apresentadas, é necessário incluir no código o seguinte cabeçalho de definições:

```
#include <omp.h>
```

Existe uma macro _OPENMP definida nas implementações, que não pode ser redefinida por um #define ou #undef. Esta macro deve ser utilizada, por exemplo, para decidir se um trecho de código deve ser compilado ou não.

```
#ifdef _OPENMP
meuid = omp_get_thread_num() + indice;
#endif
```

Modelo de programação

A região paralela é a estrutura básica de paralelismo no OpenMP. Os programas começam a execução com uma única *thread*, chamada de *thread master*. Quando a primeira região paralela, definida pela diretiva **parallel**, é encontrada, a *thread master* cria um time de *threads*, segundo o modelo *fork/join*. A Figura 6.1 ilustra esse modelo.

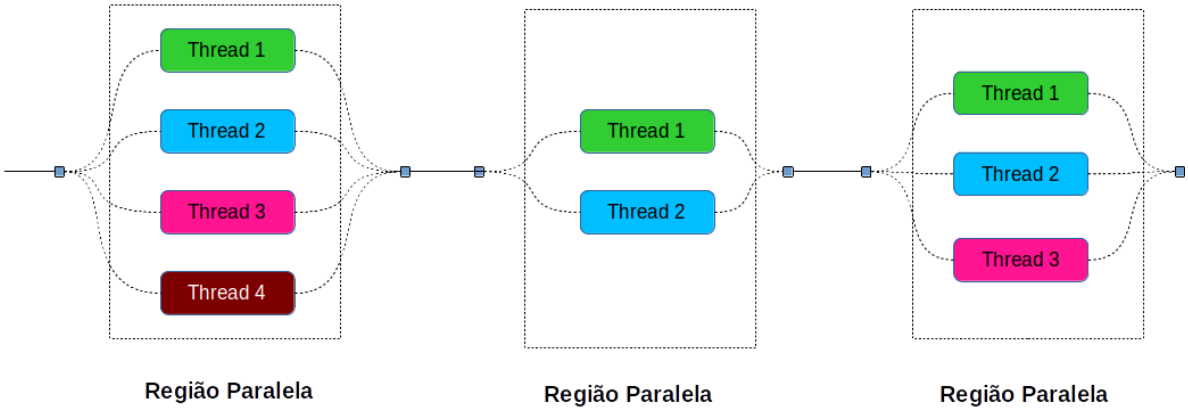

Figura 6.1: Esquema de threads do OpenMP

Cada *thread* vai executar as sentenças dentro de uma região paralela. Essa execução será em paralelo se houver um número suficiente de processadores físicos. No final da região paralela, a *thread master* espera pelo término das outras *threads*, quando prossegue a execução das outras sentenças sequencialmente.

6.2 Diretivas principais

6.2.1 Diretiva parallel

A diretiva **parallel** define uma região paralela, isto é, um trecho de código que será executado pelo time de *threads* concorrentemente. A sintaxe desta diretiva é vista a seguir:

#pragma omp parallel [cláusulas]
{
bloco paralelo
}

A diretiva **parallel** admite as cláusulas **if, num_threads, default, private, firstprivate, shared, reduction** e **copyin**, cujos detalhes serão vistos na Seção 6.4.

O exemplo listado a seguir apresenta o uso da diretiva **parallel** com a cláusula **num_threads**, para a definição do número de *threads* em execução na região paralela. Maiores detalhes são apresentados na Seção 6.4.1.

```
#include <stdio.h>
#include <omp.h>
```

```
int main(int argc, char *argv[]){   /*omp_parallel.c */
#pragma omp parallel num_threads(4)
    {
    int tid = omp_get_thread_num();
    printf ("Alo da thread %d. \n", tid);
    }
    return 0;
}
```

A função **omp_get_thread_num()** retorna o número atual da *thread* em execução, também conhecido com *thread identifier* (tid). Maiores detalhes a respeito dessa função serão vistos na Seção 6.3.

Esse programa pode ser compilado e executado com os seguintes comandos em um terminal de um sistema operacional do tipo Linux utilizando-se o compilador *gcc*:

```
$ gcc -o teste omp_parallel.c -fopenmp
$ ./teste
```

Não há necessidade de comandos especiais para a execução desse código. Um resultado possível para a execução desse código seria:

```
Alo da thread 3.
Alo da thread 0.
Alo da thread 2.
Alo da thread 1.
```

Destacamos que não há uma ordem predefinida para a execução das *threads* ou impressão dos resultados.

6.2.2 Diretiva single

A diretiva **single** define que um bloco estruturado, dentro de uma região paralela, deve ser executado por uma única *thread*. A primeira *thread* que alcançar a diretiva **single** executará o bloco. As outras *threads* do time devem esperar o final da execução do bloco, ou seja, existe uma barreira implícita no fim do bloco estruturado. A sua sintaxe é apresentada a seguir.

#pragma omp single [cláusulas]
{
bloco estruturado
}

Essa diretiva deve estar em uma região paralela especificada por uma diretiva **parallel** e deve conter um bloco estruturado, isto é, não pode haver desvio para dentro ou para fora dele. Essa diretiva pode ter cláusulas **private, firstprivate, copyprivate** e **nowait** cujos detalhes serão apresentados na Seção 6.4. Vejamos o exemplo a seguir.

```
#include <stdio.h>
#include <omp.h>
void trab1(int tid) {
     printf("Executando trab1 na thread %d \n", tid);
}
void trab2(int tid) {
    printf("Executando trab2 na thread %d \n", tid);

int main(int argc, char *argv[]) { /* omp_single.c  */
#pragma omp parallel num_threads(4)
```

```
{
    int tid =  omp_get_thread_num();
    /* Trecho executado por uma única thread, aleatoriamente */
    #pragma omp single
    trab1(tid);
    /* Trecho executado por todas as threads, sempre depois de trab1 */
    trab2(tid);
}
     return(0);
}
```

O exemplo apresentado ilustra o uso das diretivas **parallel** e **single**. Uma possível saída para a execução desse programa seria:

```
Executando trab1 na thread 2
Executando trab2 na thread 1
Executando trab2 na thread 3
Executando trab2 na thread 0
Executando trab2 na thread 2
```

6.2.3 Diretiva sections

A diretiva **sections** distribui o trabalho entre diversas *threads* limitadas a uma região paralela definida. Essa diretiva permite a execução em paralelo de blocos separados de código (por exemplo, diversas subrotinas independentes), mas não é escalável, porque a quantidade de paralelismo disponível é determinada no código-fonte, sendo portanto pouco utilizada. A diretiva **sections** indica o início de uma construção, que contém diversas diretivas **section**. Sua sintaxe é a seguinte:

```
    #pragma omp sections [cláusulas]
    {
            #pragma omp section
            {
                a ( );    // bloco estruturado
            }

            #pragma omp section
            {
                b ( );  // bloco estruturado
            }
            ...
            #pragma omp section
            {
                c ( );  // bloco estruturado
            }
            ...
    }
            ...
    d( );
```

Algumas regras para a utilização dessas diretivas são: uma diretiva **section** não pode aparecer fora do bloco estruturado abrangido pela diretiva **sections**; cada bloco estruturado, guardado por uma diretiva **section**, será executado em paralelo por uma *thread* distinta; existe uma barreira implícita definida no final da região paralela associada à diretiva **sections**, se não houver uma cláusula **nowait** especificada. Veremos o uso da cláusula **nowait** com mais detalhes na Seção 6.5.3. A Figura 6.2 procura exemplificar essa distribuição do trabalho entre as *threads* da diretiva **sections** ao longo do eixo do tempo, para o exemplo apresentado.

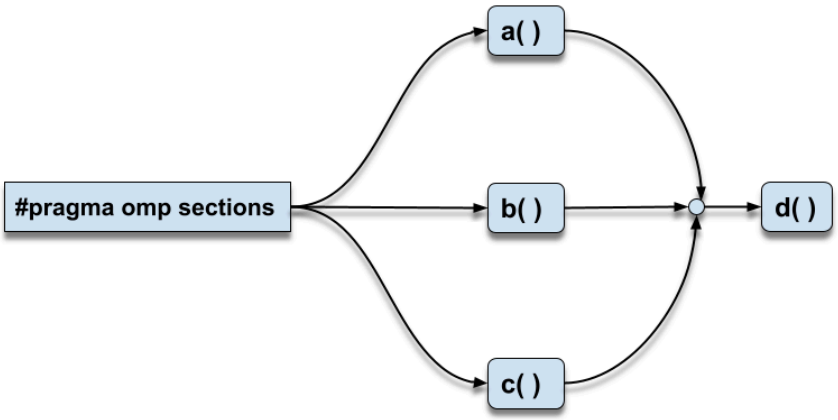

Figura 6.2: Diretiva sections

6.2.4 Diretiva parallel sections

Assim como no caso da diretiva **for** (veja a Seção [6.2.6]), a diretiva **sections** também pode ser combinada em uma única diretiva **parallel sections**.

```
#pragma omp parallel sections [cláusulas]
{
        #pragma omp section
        {
            // bloco estruturado
        }

        #pragma omp section
        {
           // bloco estruturado
        }
        ...
}
```

Ilustramos no exemplo seguinte o uso da diretiva **parallel sections**. Na realidade existe uma condição de corrida no acesso à variável global *conta_secao*, podendo resultar que um mesmo valor seja impresso nas duas seções. Vamos discutir como resolver isso em detalhes na Seção 6.5.

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) { /* omp_sections.c */
int conta_secao = 0;
#pragma omp parallel sections num_threads(4)
{
    #pragma omp section
    {
            int tid = omp_get_thread_num();
        conta_secao++;
        /* deve imprimir o número um ou dois */
        printf( "Ordem de execução da sessão %d, thread %d \n", conta_secao, tid);
    }
    #pragma omp section
    {
            int tid = omp_get_thread_num();
        conta_secao++;
```

```
        /* deve imprimir o número um ou dois */
        printf( "Ordem de execução da sessão %d, thread %d \n", conta_secao, tid );
    }
}
    return(0);
}
```

Um saída possível para a execução desse programa é a seguinte:

```
Ordem de execução da sessão 1, thread 0
Ordem de execução da sessão 2, thread 2
```

6.2.5 Diretiva for

Laços são a maior fonte de paralelismo na maioria dos códigos. A diretiva **for** permite a extração de paralelismo dos laços com a divisão da execução das iterações do laço entre as diversas *threads* de um time. Apresentaremos inicialmente apenas a forma básica dessa diretiva.

#pragma omp for [cláusulas]
laço for

Sem cláusulas adicionais, a diretiva **for** deve particionar as iterações do laço o mais igualmente possível entre as *threads*. Contudo, isso é dependente de implementação e ainda há alguma ambiguidade, por exemplo, havendo 7 iterações para serem divididas entre 3 *threads*, elas poderão ser particionadas como 3+3+1 ou 3+2+2.

A diretiva **for** possui as seguintes restrições:

- O corpo do laço deve ser um bloco estruturado, e sua execução não pode ser terminada por um comando *break*.
- Os valores para o controle das expressões do laço associados com a diretiva **for** devem ser os mesmos para todas as *threads* do laço.
- A variável de índice do laço deve ser do tipo inteiro com sinal.

Ao final do laço existe uma barreira implícita, onde todas as *threads* participantes da computação do laço devem aguardar pelas demais, para só então prosseguirem. A variável de índice do laço tem o seu valor indeterminado ao final do laço, a menos que uma cláusula **lastprivate** seja especificada. Além dessa, há diversas outras cláusulas que, por exemplo, modificam a forma como as iterações são distribuídas entre o time de *threads*. Veremos mais detalhes sobre elas na Seção 6.4.3.

6.2.6 Diretiva parallel for

A construção **for** é tão comum que existe uma forma combinada da diretiva **parallel** com a diretiva **for**:

#pragma omp parallel for [cláusulas]
laço for

O exemplo seguinte apresenta o uso desta diretiva, realizando uma operação *multiply-add* ou SAXPY, que significa "Single-Precision A·X Plus Y". Essa é uma função básica de bibliotecas como a *Basic Linear Algebra Subroutines* (BLAS), e consiste da multiplicação de um vetor por um valor que

é somado a uma variável ou constante. A diretiva **parallel for** pode usar todas as cláusulas disponíveis para a diretiva **parallel** e para a diretiva **for**, como veremos mais adiante.

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
    #pragma omp parallel num_threads(4)
    {
    int tid = omp_get_thread_num();
    #pragma omp for
    for (int i = 0; i < n; ++i)
    {
        y[i] = a*x[i] + y[i];
        printf("thread %d i = %d \n", tid,i);
    }
   }
}

int main(int argc, char *argv[]) { /* omp_parfor.c  */
int n = 20; // 20 floats
float *x = (float*)malloc(n*sizeof(float));
float *y = (float*)malloc(n*sizeof(float));
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    free(x);
    free(y);
    return(0);
}
```

A saída do programa ilustra a distribuição das iterações do laço entre as *threads*, com o mesmo número de iterações alocado para cada *thread*, na mesma ordem que o seu *tid*. Essa distribuição pode ser alterada com cláusulas específicas de escalonamento que serão apresentadas na Seção 6.4.3.

```
thread 0 i = 0
thread 0 i = 1
thread 0 i = 2
thread 0 i = 3
thread 0 i = 4
thread 3 i = 15
thread 3 i = 16
thread 3 i = 17
thread 3 i = 18
thread 3 i = 19
thread 2 i = 10
thread 2 i = 11
thread 2 i = 12
thread 2 i = 13
thread 2 i = 14
thread 1 i = 5
thread 1 i = 6
thread 1 i = 7
thread 1 i = 8
thread 1 i = 9
```

6.3 Funções OPENMP

O comportamento de um programa pode ser definido por algumas variáveis de ambiente e funções da biblioteca OpenMP. A seguir apresentamos algumas dessas funções utilizadas pelo OpenMP e suas aplicações.

6.3.1 Número de threads na região paralela

O número de *threads* que serão disparadas em uma região paralela pode ser especificado de diversas maneiras.

- Variável de ambiente OMP_NUM_THREADS
- Função omp_set_num_threads(n)
- Cláusula num_threads(n) (Veja a Seção 6.4)

Variável de ambiente OMP_NUM_THREADS

Na primeira delas, fora do programa, o número de *threads* pode ser definido com o uso da variável de ambiente OMP_NUM_THREADS, conforme um dos comandos a seguir, dependendo da sua versão de sistema operacional, em um terminal de um sistema operacional tipo Linux, onde Math input error é um valor numérico.

```
$ export OMP_NUM_THREADS=n
```

ou

```
$ setenv OMP_NUM_THREADS n
```

A variável de ambiente OMP_NUM_THREADS, quando presente, especifica inicialmente o número de *threads* usadas durante a execução de regiões paralelas. O valor padrão para esta variável é 1. Serão usadas então OMP_NUM_THREADS *threads* para executar o programa, independente do número de processadores físicos disponíveis no sistema.

Como resultado, podem ser executados programas com um número de *threads* maior do que o número de processadores físicos. Esses programas serão executados corretamente, contudo, o ganho de desempenho na execução dos programas desta maneira não será significativo.

Função omp_set_num_threads

A função **omp_set_num_threads** tem sua sintaxe definida como a seguir:

void omp_set_num_threads(int num_threads);

As chamadas para a função **omp_set_num_threads()** se sobrepõem ao valor especificado na variável de ambiente OMP_NUM_THREADS. Essa função deve ser chamada fora de uma região paralela, porque a mudança do número de *threads* dentro de uma região paralela não é permitida. Se por acaso isso for feito, o comportamento do programa não pode ser determinado.

Há um outro conjunto de funções utilizadas para fornecer informações como o número de *threads* em execução na região paralela, o número de identificação da *thread*, entre outras, como veremos a seguir.

Função omp_get_num_threads

A função **omp_get_num_threads** é muito utilizada no OpenMP. Ela retorna um valor inteiro com o número de *threads* em execução na região paralela.

Nota importante: esta função retorna o valor '1' se a chamada for feita fora de uma região paralela.

int omp_get_num_threads(void);

Função omp_get_thread_num

A função **omp_get_thread_num** serve para determinar o número atual da *thread* em execução, também chamado de *thread identifier* (tid). As *threads* do OpenMP são numeradas com valores entre 0 e ***omp_get_num_threads()-1***. A sua sintaxe é a seguinte:

int omp_get_thread_num(void);

O número de *threads* em execução fora da região paralela é sempre igual a '1'. Apresentamos no próximo exemplo o uso dessas três funções.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) { /* omp_threadnum.c  */
   omp_set_num_threads(4);
   printf("Número de threads fora da região paralela = %d\n", omp_get_num_threads());
#pragma omp parallel
{
int nthreads, tid;
    /* Obtém o número da thread */
    tid = omp_get_thread_num();
    printf("Alô da thread = %d\n", tid);
    /* Apenas a thread master faz isto */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Número de threads na região paralela= %d\n", nthreads);
    }
}   /* Todas as threads se juntam à thread master e terminam */
    return(0);
}
```

Uma saída possível para a execução do programa, após sua compilação, seria a seguinte:

```
Número de threads fora da região paralela = 1
Alô da thread = 0
Número de threads na região paralela= 4
Alô da thread = 2
Alô da thread = 1
Alô da thread = 3
```

Função omp_get_num_procs

Essa função retorna o número de processadores disponíveis para a execução do programa no momento da chamada. O seu formato é o seguinte:

int omp_get_num_procs(void);

Função omp_in_parallel

Existe ainda uma função para identificar se a execução atual é dentro de uma região paralela ou não. Normalmente tem sentido quando chamado dentro de uma subrotina para verificar se ela foi invocada dentro de uma região paralela ou não.

int omp_in_parallel(void);

Nota: Esta função retorna um valor diferente de 0 se chamada de dentro de uma região paralela.

A rotina **omp_get_max_threads** retorna o limite superior para o número de threads que poderiam ser utilizadas para formar um novo time de *threads* se uma construção paralela sem a cláusula **num_threads** (veja a Seção 6.4) for encontrada depois da execução da chamada dessa rotina. Normalmente é retornado o valor definido na variável de ambiente OMP_NUM_THREADS ou pela função **omp_set_num_threads()**.

int omp_get_max_threads(void);

6.3.2 Medindo o tempo de execução

A função **omp_get_wtime()** é utilizada no OpenMP para retornar o tempo decorrido, em segundos, a partir de um mesmo ponto do início da execução do programa.

double omp_get_wtime (void);

Assim, basta realizarmos a chamada uma vez no início e outra vez no final do programa, e a diferença entre esses valores informa o tempo gasto em segundos para execução do trecho de programa em questão. Essa função retorna uma variável do tipo *double* como resultado.

Há ainda uma outra rotina, menos utilizada, que retorna a precisão do relógio, expressa pelo número de segundos entre dois tiques sucessivos do relógio. Assim consegue-se determinar a precisão das medidas de tempo realizadas.

double omp_get_wtick(void);

Apresentamos no próximo exemplo o uso dessas funções.

```
int main (int argc, char *argv[]) { /* omp_numprocs.c  */
double inicio, fim;
   inicio = omp_get_wtime();
/* Imprime se estiver na região paralela */
   if (omp_in_parallel())
    printf("Dentro da região paralela \n");
   else
    printf("Fora da região paralela \n");

#pragma omp parallel
{
int num_procs, max_threads, tid;
    tid = omp_get_thread_num();
/* Apenas a thread master faz isto */
    if (tid == 0) {
        num_procs = omp_get_num_procs();
        max_threads = omp_get_max_threads();
        printf("Número de processadores disponíveis = %d\n", num_procs);
```

```
/* Imprime o número máximo de threads */
        printf("Número máximo de threads = %d\n", max_threads);
/* Imprime se estiver na região paralela */
   if (omp_in_parallel())
    printf("Dentro da região paralela \n");
   else
    printf("Fora da região paralela \n");
    }
} /* Todas as threads se juntam à thread master e terminam */
    fim = omp_get_wtime();
    printf("Tempo gasto na execução = %3.6f segundos. \n", fim-inicio);
    printf("Precisão da medida = %3.9f segundos\n", omp_get_wtick());
    return(0);
}
```

Uma saída possível para a execução do programa, após sua compilação, seria a seguinte:

```
Fora da região paralela
Número de processadores disponíveis = 8
Número máximo de threads = 20
Dentro da região paralela
Tempo gasto na execução = 0.000708 segundos.
Precisão da medida = 0.000000001 segundos
```

6.4 Cláusulas

As cláusulas especificam informação adicional para uma diretiva na região paralela, como:

```
#pragma omp parallel [cláusulas]
#pragma omp parallel for [cláusulas]
#pragma omp parallel sections [cláusulas]
```

As cláusulas são separadas por vírgula ou espaço no Fortran, e por espaço no C/C++.

6.4.1 Cláusulas da diretiva parallel

A diretiva **parallel** admite as seguintes cláusulas:

1. **num_threads**
2. **default, private, shared**
3. **firstprivate**
4. **reduction**
5. **if**
6. **copyin** (não abordado)

Vejamos a seguir como utilizar algumas dessas cláusulas.

Cláusula num_threads

O uso da cláusula **num_threads** é uma outra opção para especificar o número de *threads* para a execução de uma região paralela. Essa cláusula pode ser especificada em conjunto com cada uma das seguintes diretivas OpenMP:

- **parallel**
- **parallel for**

- **parallel sections**

A presença da cláusula **num_threads()** em uma região paralela se sobrepõe às outras formas apresentadas anteriormente (OMP_NUM_THREADS ou pela função **omp_set_num_threads()**). Veremos no próximo exemplo o uso da cláusula **num_threads** com a diretiva **parallel**.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) {  /* omp_numthreads.c  */
   printf("Fora = %d\n", omp_in_parallel( ));
   printf("Número de processadores = %d\n", omp_get_num_procs( ));
   #pragma omp parallel num_threads(3)
   {
       int tid = omp_get_thread_num();
       printf("Olá da thread %d\n", tid);
       printf("Dentro = %d\n", omp_in_parallel( ));
   }
   return(0);
}
```

Após ser compilado, um possível resultado da execução desse programa seria o seguinte:

```
Fora = 0
Número de processadores = 4
Olá da thread 0
Dentro = 1
Olá da thread 2
Dentro = 1
Olá da thread 1
Dentro = 1
```

Cláusulas shared, private e default

Dentro de uma região paralela, as variáveis podem ser privadas ou compartilhadas. As variáveis compartilhadas, cujo conteúdo é o mesmo em todas as *threads*, podem ser lidas ou escritas a partir de qualquer *thread*. Por outro lado, cada *thread* tem a sua própria cópia de variáveis privadas: essas variáveis, embora possam ter o mesmo nome, são invisíveis para as outras *threads* e uma variável privada pode ser lida ou escrita apenas pela sua própria *thread*.

- **shared(lista-de-variáveis)**
- **private(lista-de-variáveis)**
- **default(shared|none)**

A cláusula **shared** serve para declarar as variáveis listadas em *lista-de-variáveis* como compartilhadas, assim como a cláusula **private** serve para declará-las como privadas. A cláusula **default** tem dois parâmetros possíveis: **shared** e **none**. Ao se especificar **default(shared)**, as variáveis não listadas serão compartilhadas por padrão. Contudo, ao se especificar **default(none)**, a situação é um pouco mais complicada. O uso dessa cláusula obriga que, para as variáveis referenciadas no contexto de uma construção paralela, pelo menos uma das seguintes condições seja verdadeira:

- A variável esteja explicitamente listada na cláusula **shared**;
- A variável seja declarada dentro da região paralela;
- A variável tenha um tipo que a define como uma constante;
- A variável seja o índice do laço mais externo de uma diretiva **for** ou **parallel for**;
- A variável seja declarada como **threadprivate** (não abordado neste livro).

Em caso contrário, o compilador emitirá uma mensagem de erro, indicando que a variável não foi especificada como **shared** ou **private**. A ausência de uma cláusula explícita na diretiva **parallel** tem o comportamento padrão de uma cláusula **default shared**. Mas como decidir se uma variável deve ser compartilhada ou privada?

- A maioria das variáveis são compartilhadas;
- Os índices dos laços são privados;
- As variáveis temporárias dos laços são compartilhadas;
- As variáveis apenas de leitura são compartilhadas;
- As matrizes principais são compartilhadas;

O uso da cláusula **shared** é ilustrado no código a seguir. Note que a variável de índice do laço ("i") não precisa ser declarada como **private**, já que é sempre privada por padrão.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char *argv[]) { /* omp_shared.c  */
int i, n = 7;
int a[n];
    (void) omp_set_num_threads(4);
    for (i = 0; i < n; i++)
        a[i] = i+1;
    #pragma omp parallel for shared(a,n) default(none)
    for (i = 0; i < n; i++) {
        a[i] += i;
    } /*-- Fim do parallel for --*/
    printf("No programa principal depois do parallel for:\n");
    for (i = 0; i < n; i++)
        printf("a[%d] = %d\n",i,a[i]);
    return(0);
}
```

Como o vetor "a" é compartilhado entre todas as *threads*, a saída desse programa, que é sempre executado com quatro *threads*, será a seguinte:

```
No programa principal depois do parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
```

No código a seguir encontramos um exemplo do uso da cláusula **private**.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char *argv[]) { /* omp_private.c  */
int i, n = 5, a=1000;
    (void) omp_set_num_threads(3);
    #pragma omp parallel for private(a)
    for (i = 0; i < n; i++) {
        a = i+1;
                printf("Thread %d tem um valor de a = %d para i = %d\n",
omp_get_thread_num(),a,i);
    }
```

```
    return(0);
}
```

Novamente, a variável "i" não precisa ser declarada como privada, porque a variável do índice do laço mais externo é sempre privada por padrão.

A variável "a" é declarada como privada, e tem um valor distinto em cada *thread* e diferente do valor da *thread master*, desse modo a saída do programa será a seguinte.

```
Thread 0 tem um valor de a = 1 para i = 0
Thread 0 tem um valor de a = 2 para i = 1
Thread 2 tem um valor de a = 5 para i = 4
Thread 1 tem um valor de a = 3 para i = 2
Thread 1 tem um valor de a = 4 para i = 3
```

Cláusula firstprivate

As variáveis privadas não têm valor inicial no início da região paralela. Para se atribuir um valor inicial deve-se utilizar a cláusula **firstprivate**, que copia o valor atribuído à variável na *thread master*, fora da região paralela, para cada uma das *threads* do time dentro da região paralela. A sua sintaxe é a seguinte:

firstprivate(lista-de-variáveis)

Vemos a seguir um exemplo com o uso desta cláusula, acompanhado de uma possível saída para sua execução.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]) { /* omp_firstprivate2.c  */
int a=1000, n = 5;
    omp_set_num_threads(3);
    #pragma omp parallel firstprivate(a)
    {
    int tid = omp_get_thread_num();
    printf("Thread %d recebe o valor inicial de a = %d \n", tid,a);
        #pragma omp for
        for (int i = 0; i < n; i++) {
            a += i;
        }
    printf("Thread %d tem o valor final de a = %d \n", tid,a);
    }
    printf("Thread master tem um valor de a = %d  \n", a);
    return(0);
}
```

```
Thread 0 recebe o valor inicial de a = 1000
Thread 1 recebe o valor inicial de a = 1000
Thread 2 recebe o valor inicial de a = 1000
Thread 2 tem o valor final de a = 1004
Thread 1 tem o valor final de a = 1005
Thread 0 tem o valor final de a = 1001
Thread master tem um valor de a = 1000
```

Note que todas as *threads* dentro da região paralela recebem o mesmo valor inicial para a variável "a", no caso, 1000. Em seguida, as iterações do laço são divididas entre as *threads* e cada uma

calcula um valor final diferente para a variável privada "a". Aliás, você pode inferir a distribuição das iterações do laço para cada uma das *threads* com base nos resultados obtidos?

Como a variável "a" é privada, o valor calculado pelo time de *threads* no laço não é transmitido, ao final da região paralela, para a variável "a" na *thread master*. Você precisa estar atento ao escopo de cada variável, pois o fato de terem o mesmo nome não significa que sejam a mesma variável na memória. Os casos de uso da cláusula **firstprivate** são surpreendentemente raros.

Cláusula reduction

Tabela 6.1: Operadores de redução do OpenMP

operador	operação	valor inicial
+	soma	0
-	subtração	0
*	multiplicação	1
&	"e" bit a bit	~0
|	"ou" bit a bit	0
^	"ou exclusivo" bit a bit	0
&&	"e" lógico	1
||	"ou" lógico	0
max	máximo	menor repr.
min	mínimo	maior repr.

Uma redução produz um único valor de uma variável, cujo valor é local nas diversas *threads*, a partir de operações tais como soma, multiplicação, máximo, mínimo, "e" e "ou". Vejamos o exemplo apresentado seguir:

```
b = 0;
for (i = 0; i < n; i++)
     b += a[i];
```

Se a variável "b" fosse compartilhada por todas as *threads*, e se fosse permitido o acesso apenas de uma *thread* por vez, não haveria paralelismo na execução do programa. Ao invés disso, cada *thread* pode acumular uma soma parcial em sua própria cópia privada, e então essas cópias são reduzidas para se obter o resultado final na *thread master*. A sintaxe de uso da cláusula **reduction** é mostrada a seguir:

reduction(operador:lista-de-variáveis)

Os operadores válidos para a cláusula **reduction** podem ser vistos na Tabela 6.1. O exemplo listado em seguida mostra o uso da cláusula **reduction** em conjunto com a diretiva **parallel for**. Neste exemplo é realizada a soma, em paralelo pelo time de *threads*, de todos os elementos do vetor "a".

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SOMA_INICIAL 1000
int main(int argc, char *argv[]) { /* omp_reduction.c  */
int i, n = 25;
int soma, a[n];
int ref = SOMA_INICIAL + ((n-1)*n)/2;
```

```
    omp_set_num_threads(4);
    for (i = 0; i < n; i++)
        a[i] = i;
    soma = SOMA_INICIAL;
    printf("Valor da soma antes da região paralela: %d \n",soma);
    #pragma omp parallel default(none) shared(n,a) reduction(+:soma)
    {
    soma=0;
    #pragma omp for
    for (i = 0; i < n; i++)
        soma += a[i];
    printf("Valor da soma na thread: %d é igual %d\n",omp_get_thread_num(), soma
);
    } /*-- Fim da redução paralela --*/
    printf("Valor da soma depois da região paralela: %d\n",soma);
    printf("Verificação do resultado: soma = %d (deveria ser %d)\n",soma,ref);
    return(0);
}
```

Como resultado da execução do programa, teremos a saída apresentada a seguir:

```
Valor da soma antes da região paralela: 1000
Valor da soma na thread: 3 é igual 129
Valor da soma na thread: 0 é igual 21
Valor da soma na thread: 2 é igual 93
Valor da soma na thread: 1 é igual 57
Valor da soma depois da região paralela: 1300
Verificação do resultado: soma = 1300 (deveria ser 1300)
```

As variáveis de redução são atualizadas, combinando o valor armazenado na *thread master* com as cópias privadas das demais *threads*, ao final da região paralela onde a cláusula de redução foi especificada. A partir da versão 4.5 do OpenMP, os arranjos também podem ser usados como variáveis de redução (anteriormente isso só era possível com escalares e elementos de um arranjo), como apresentado a seguir.

```
    #pragma omp parallel for private (i) reduction (+:b)
         for (j = 0; j < N; j++)
              for (i = 0; i < M; i++)
                   b(i) = b(i) + b(i,j);
```

O tipo de uma variável listada em uma cláusula de redução deve ser um tipo válido para o operador de redução utilizado. Por exemplo, para um operador de redução *max* ou *min* em C, o item deve ser do tipo *char, int, float, double* ou *_Bool*, possivelmente modificado com *long, short, signed* ou *unsigned*.

Cláusula if

Sintaxe:

if (expressao)

Onde *expressao* é uma expressão inteira que, se avaliada como verdadeira (não nula), habilita a execução em paralelo do código na região. Se a expressão for avaliada como falsa (zero) a região é executada sequencialmente (por uma única *thread*). Ilustramos a seguir o uso dessa cláusula, acompanhado de uma possível saída para sua execução.

```
#include <stdio.h>
#include <omp.h>
void teste(int valor)  {
#pragma omp parallel if (valor)
    if (omp_in_parallel()) {
        #pragma omp single
        printf("Valor = %d, paralelizada com %d threads\n", valor, omp_get_num_threads());
    } else {
        printf("Valor = %d, serializada\n", valor);
    }
}
int main(int argc, char *argv[]) {  /* omp_if.c  */
    omp_set_num_threads(2);
    teste(0);
    teste(2);
    return(0);
}

Valor = 0, serializada
Valor = 2, paralelizada com 2 threads
```

6.4.2 Cláusulas da diretiva single

A diretiva pode ter cláusulas **private, firstprivate, copyprivate e nowait**. As diretivas **private** e **firsprivate** têm comportamento idêntico àquele da diretiva **parallel**.

Além disso, temos as seguintes restrições para uso de cláusulas com a diretiva **single**: apenas uma cláusula **nowait** pode aparecer na diretiva **single**; a cláusula **copyprivate** não deve ser utilizada em conjunto com a cláusula **nowait**. A cláusula **nowait** serve para remover a barreira implícita ao final do bloco estruturado associado à diretiva **single**. Maiores detalhes serão vistos na Seção 6.5.3.

Cláusula copyprivate

A cláusula **copyprivate** faz a difusão do valor atribuído a uma variável privada de uma *thread* diretamente para todas as instâncias da variável privada nas outras *threads*. É uma alternativa ao uso de uma variável compartilhada, quando isso for difícil, por exemplo, em funções recursivas requerendo diferentes variáveis em cada nível. A cláusula **copyprivate** só pode ser utilizada com a diretiva **single**. A sintaxe da cláusula **copyprivate** é a seguinte:

copyprivate(lista-de-variáveis)

O valor das variáveis em *lista-de-variáveis* é calculado pela *thread* que executou o bloco estruturado associado com a construção **single**. Esse valor é copiado para as demais *threads* antes que ultrapassem a barreira implícita no final da construção **single**. As variáveis especificadas na cláusula **copyprivate** não podem aparecer em uma cláusula **private** ou **firstprivate** para a mesma diretiva **single**. Um exemplo de uso dessa cláusula é apresentado a seguir, acompanhado de um possível resultado da execução deste programa.

```
#include <stdio.h>
#include <stdlib.h>
float   read_next( ) {
float*  tmp;
float   return_val;
#pragma omp single copyprivate(tmp)
    {  /* uma thread aloca espaço para a variável */
```

```
        tmp = (float*) malloc(sizeof(float));
    }  /* o endereço armazenado no ponteiro é copiado  */
#pragma omp master
    { /* a thread master faz a leitura do valor */
        scanf("%f", tmp);
    }
#pragma omp barrier
     /* todas as threads recebem o valor lido */
    return_val = *tmp;
#pragma omp barrier
#pragma omp single
    {
        free(tmp);
    }  /* apenas uma thread libera o espaço alocado */
    return return_val;
}
int main(int argc, char *argv[]) { /* omp_copyprivate.c  */
   float a;
   #pragma omp parallel
   {
       a = read_next( );
       printf("Valor lido %.3f \n", a);
   }
}
```

```
98.765
Valor lido 98.765
Valor lido 98.765
Valor lido 98.765
Valor lido 98.765
```

6.4.3 Cláusulas da diretiva for

Laços são a maior fonte de paralelismo na maioria dos códigos. Diretivas paralelas de laços são portanto muito importantes! Uma diretiva **parallel for** divide as iterações do laço entre as *threads*.

#pragma omp for [cláusulas]
laço for

Como já dito, sem cláusulas adicionais, a diretiva **for** usualmente particionará as iterações o mais igualmente possível entre as *threads*. Mas como saber se um laço pode ser executado em paralelo ou não? A resposta é simples, verificamos se o laço dá o mesmo resultado se executado na ordem inversa, isto é, com o índice do laço variando do maior para o menor valor. Se for o caso, então ele é quase certamente paralelizável. Desvios para fora de um laço paralelo não são permitidos. Vejamos alguns exemplos:

```
Laço 1:
    for (i = 1; i < n; i++) {
         a[i] = 2*a[i-1];
    }

Laço 2:
    ix = base;
    for (i = 0; i < n; i++) {
         a[ix] = a[ix]* b[i];
         ix = ix + stride;
    }
Laço 3:
    for (i = 0; i < n; i++) {
```

```
        b[i]= (a[i] - a[i-1])*0.5;
    }
```

No exemplo apresentado, os Laços 1 e 2 não são paralelizáveis, porque apresentam resultados diferentes quando o sentido da iteração do laço é invertido. Já o Laço 3 pode ser paralelizado conforme exemplificado a seguir.

```
    #pragma omp parallel for
        for (i = 1; i <= n; i++) {
            b[i] = (a[i]- a[i-1])*0.5;
    }
    /* Fim do parallel for */
```

A diretiva **for** pode ter cláusulas **private e firstprivate** se referindo às variáveis no escopo do laço. A cláusula **nowait** remove a barreira implícita ao final do laço e será vista em detalhes na Seção 6.5.3. Observe que a variável de índice do laço mais externo é privada por padrão, ao contrário dos outros índices de laços internos, que são compartilhados caso nada seja declarado. A diretiva **parallel for** pode usar todas as cláusulas disponíveis para a diretiva **parallel**. Relembrando:

- **num_threads**
- **default, private, shared**
- **reduction**
- **firstprivate**
- **if**
- **copyin** (não abordado)

Além dessas, estão definidas as seguintes cláusulas específicas para a diretiva **for**: **lastprivate**, **schedule**, **ordered** e **collapse**, cujas sintaxes e aplicações veremos em detalhes a seguir.

Cláusula lastprivate

O valor que uma variável privada tem na saída de um laço é normalmente indefinido. Deve-se utilizar a cláusula **lastprivate** para que esse valor seja idêntico àquele da execução sequencial do laço.

lastprivate(lista-de-variáveis)

Esta cláusula também se aplica à diretiva **sections**, quando a variável terá um valor igual àquele atribuído na última seção executada (veja a Seção 6.4.4). Apresentamos o uso dessa cláusula no próximo exemplo.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char *argv[]) { /* omp_lastprivate.c  */
int i;
int x = 33;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i <= 10; i++) {
        x = i;
        printf("Número da thread: %d x: %d\n",omp_get_thread_num(),x);
    }
    printf("x é %d\n", x);
    return(0);
}
```

Um possível resultado da execução seria:

```
Número da thread: 0    x: 0
Número da thread: 0    x: 1
Número da thread: 0    x: 2
Número da thread: 3    x: 9
Número da thread: 3    x: 10
Número da thread: 2    x: 6
Número da thread: 2    x: 7
Número da thread: 2    x: 8
Número da thread: 1    x: 3
Número da thread: 1    x: 4
Número da thread: 1    x: 5
x é 10
```

Cláusula schedule

A cláusula **schedule** permite uma variedade de opções para especificar quais *threads* executarão as iterações dos laços.

schedule (tipo[, tam])

onde **tipo** pode ser **static, dynamic, guided, auto** ou **runtime** e *tam* é uma expressão inteira com valor positivo. No escalonamento **static**, sem a especificação de *tam*, o espaço de iteração é dividido em pedaços (aproximadamente) iguais e cada pedaço é atribuído a cada *thread* (escalonamento em bloco). Se o valor de *tam* for especificado, o espaço de iteração é dividido em pedaços, cada um com *tam* iterações, e os pedaços são atribuídos ciclicamente a cada *thread* (escalonamento cíclico por bloco).

Figura 6.3: Escalonamento Estático

O escalonamento **dynamic** divide o espaço de iteração em pedaços com *tam* iterações, e os atribui para as *threads* com uma política "primeiro a chegar, primeiro a ser atendido", isto é, se uma *thread* terminou um pedaço, ela receberá o próximo pedaço da lista. Quando nenhum valor de *tam* for especificado, o valor padrão assumido é '1'.

Figura 6.4: Escalonamento Dinâmico

O escalonamento **guided** é similar ao **dynamic**, mas os pedaços iniciam grandes e se tornam menores exponencialmente. O tamanho do próximo pedaço é (com arredondamento para cima) o número de iterações restantes dividido pelo número de *threads*. O valor *tam* especifica o tamanho mínimo dos pedaços, exceto para o pedaço com as últimas iterações, que pode ser menor. Quando nenhum valor de *tam* for especificado, o padrão é igual a '1'.

Figura 6.5: Escalonamento Guided

Por exemplo, se tivermos 50 iterações e 4 *threads*, as iterações serão divididas da forma apresentada a seguir entre as *threads*, lembrando que a alocação exata para cada *thread* é

realizada de modo dinâmico.

```
T1 =           50 / 4 = 12,5 --> 13
T2 = 50 - 13 = 37 / 4 = 9,25 --> 10
T3 = 37 - 10 = 27 / 4 = 6,75 --> 7
T4 = 27 - 7  = 20 / 4 = 5    --> 5
T4 = 20 - 5  = 15 / 4 = 3,75 --> 4
T2 = 15 - 4  = 11 / 4 = 2,75 --> 3
T1 = 11 - 3  = 8 / 4  = 2    --> 2
T3 = 8 - 2   = 6 / 4  = 1,5  --> 2
T1 = 6 - 2   = 4 / 4  = 1    --> 1
T3 =                           1
T4 =                           1
T2 =                           1
```

O escalonamento **auto** dá liberdade para o ambiente de execução escolher qual método de escalonamento será utilizado. Isso pode ser útil, por exemplo, quando o laço paralelo é executado diversas vezes, permitindo ao ambiente de execução convergir para um tipo de escalonamento com um bom balanceamento de carga e com baixa sobrecarga.

O escalonamento **runtime** delega a escolha do escalonamento para o momento da execução do programa, podendo ser determinado pelo valor da variável de ambiente OMP_SCHEDULE ou pela chamada à função **omp_set_schedule()**.

Quando presente, a variável de ambiente OMP_SCHEDULE especifica o tipo de escalonamento a ser utilizado nos laços. Esses tipos podem ser somente **static**, **dynamic**, **guided** ou **auto**.

```
$ export OMP_SCHEDULE=tipo[,tam]
```

Sendo uma variável de ambiente, sua definição é feita com o comando a seguir em um terminal de um sistema operacional tipo Linux.

```
$ export OMP_SCHEDULE="guided,4"
$ export OMP_SCHEDULE="static,1000"
```

Ou dependendo do tipo de distribuição do seu sistema operacional:

```
$ setenv OMP_SCHEDULE "static,5"
$ setenv OMP_SCHEDULE "dynamic"
```

A função **omp_set_schedule()** serve tanto para escolher algum tipo padrão de escalonamento definido pelo OpenMP quanto um escalonamento definido pela implementação do OpenMP. A chamada dessa função também se sobrepõe aos valores definidos pela variável de ambiente OMP_SCHEDULE. A sintaxe dessa função é definida da seguinte forma:

void omp_set_schedule(omp_sched_t tipo, int tam);

O primeiro parâmetro *tipo* define o tipo do escalonamento e pode ser uma das constantes listadas a seguir. O segundo parâmetro *tam* indica qual o tamanho de cada pedaço a ser enviado para cada uma das *threads*.

- omp_sched_static - escalonamento estático
- omp_sched_dynamic - escalonamento dinâmico
- omp_sched_guided - escalonamento guiado
- omp_sched_auto - escalonamento automático

Algumas restrições comportamentais se aplicam na combinação dos parâmetros utilizados nesta função. Por exemplo, se o valor indicado como tamanho de cada pedaço for menor que 1, esse

valor será definido automaticamente para o valor padrão. Outro exemplo é quando o escalonamento escolhido for o *omp_sched_auto*, que desconsidera qualquer valor informado com o parâmetro *tam*.

Qual tipo de escalonamento deve ser utilizado?

- **static**: melhor para laços balanceados – menor sobrecarga;
- **static,n**: melhor para laços com desbalanceamento suave;
- **dynamic**: útil se as iterações têm grande variação de carga, mas acaba com a localidade espacial dos dados e tem o custo adicional de troca de *threads*;
- **guided**: frequentemente menos cara que **dynamic**, mas tenha cuidado com laços onde as primeiras iterações são as mais demoradas;
- **auto**: pode ser útil se o laço é executado diversas vezes ao longo da execução do programa;
- Use **runtime** para uma experimentação adequada. É ilegal especificar um valor de *tam* com o escalonamento **runtime**.

Vejamos no próximo exemplo o uso da cláusula **schedule** com a opção **static**, com ou sem o parâmetro *tam* definido. Esse código mostra com clareza como é feita a distribuição das iterações entre as *threads*. Esse exemplo pode ser modificado para visualizar a divisão da iteração entre as *threads* com o uso das opções **dynamic** e **guided**.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[]) { /* omp_schedule.c  */
int n = 20;

    printf("Clausula static SEM o parametro tam \n");
#pragma omp parallel for schedule (static) num_threads(4)
    for (int i = 0; i < n; ++i) {
        printf("tid =%d iteracao = %d \n", omp_get_thread_num(),i);
    }

    printf("Clausula static COM o parametro tam \n");
#pragma omp parallel for schedule (static,3) num_threads(4)
    for (int i = 0; i < n; ++i) {
        printf("tid =%d iteracao = %d \n", omp_get_thread_num(),i);
    }

    return 0;
}
```

Como a saída do programa é muito extensa, ela foi omitida no texto, mas basicamente apresenta a distribuição das iterações entre as *threads*, com ou sem o uso do parâmetro Math input error na cláusula **schedule**.

Cláusula ordered

A cláusula **ordered** deve ser utilizada com a diretiva **for**, caso a diretiva **ordered** seja utilizada dentro do corpo do laço.

ordered

Diretiva ordered

A diretiva **ordered** pode ser utilizada dentro de uma região de uma diretiva **for** que tenha uma cláusula **ordered** especificada. Essa diretiva serve para indicar que um trecho de código dentro do laço deverá ser executado na mesma ordem na qual seria executado sequencialmente. A sua sintaxe é a seguinte:

#pragma omp ordered
bloco estruturado

No exemplo a seguir podemos ver o uso da cláusula e da diretiva **ordered**:

```
#include <stdio.h>
#include <omp.h>
void teste(int first, int last)
{
    #pragma omp for schedule(static) ordered
    for (int i = first; i <= last; ++i) {
         // Faz alguma coisa aqui
         if (i % 2)  {
            #pragma omp ordered
            printf("teste() iteração %d\n", i);
         }
    }
}
void teste2(int iter)
{
    #pragma omp ordered
    printf("teste2() iteração %d\n", iter);
}
int main(int argc, char *argv[]) { /* omp_ordered.c  */
// Faz alguma coisa aqui
int i;
#pragma omp parallel
    {
    teste(1, 8);
    #pragma omp for ordered
    for (i = 0 ; i < 5 ; i++)
         teste2(i);
    }
    return(0);
}
```

Um resultado possível para a execução desse programa seria:

```
teste() iteração 1
teste() iteração 3
teste() iteração 5
teste() iteração 7
teste2() iteração 0
teste2() iteração 1
teste2() iteração 2
teste2() iteração 3
teste2() iteração 4
```

A **diretiva ordered** não pode ser empregada sem a **cláusula ordered** correspondente, ou seja, não se pode utilizar uma diretiva **ordered** dentro de um laço **for** sem uma cláusula **ordered**.

Cláusula collapse

A cláusula **collapse** é utilizada para aumentar o número total de iterações que serão particionadas entre as *threads* disponíveis para execução, aumentando a granularidade do trabalho a ser realizado por cada *thread*. Se a quantidade de trabalho a ser realizado por cada *thread* for significativa (após a aplicação da compactação), isso poderá melhorar a escalabilidade paralela programa. Vejamos a seguir um exemplo do seu uso.

```
#include <stdio.h>
int main(int argc, char *argv[]) { /* omp_collapse.c  */
int j, k, jlast, klast;
#pragma omp parallel
    {
    #pragma omp for collapse(2) lastprivate(jlast, klast)
    for (k = 1; k <= 2; k++)
         for (j = 1; j <= 3; j++) {
              jlast = j;
              klast = k;
         }
    #pragma omp single
    printf("%d %d\n", klast, jlast);
    }
}
```

No exemplo apresentado as variáveis de índice Math input error e Math input error são associadas à diretiva **loop**. Logo, as iterações dos laços com as variáveis Math input error e Math input error serão agregadas em um laço com um espaço de iteração maior, e o novo laço será então dividido entre as *threads* do time atual. A ordem sequencial das iterações nos laços com as variáveis Math input error e Math input error determina a ordem das iterações no novo laço colapsado. Desse modo, o valor final das variáveis Math input error e Math input error no novo laço colapsado será 2 e 3, respectivamente. Como as variáveis Math input error e Math input error são declaradas como **lastprivate**, seus valores finais serão também 2 e 3, respectivamente. Como resultado da execução desse programa poderíamos ter:

```
2 3
```

Pode-se melhorar o desempenho do programa evitando o uso dos índices dos laços colapsados (se possível) no interior do novo laço resultante. Isso evita que o compilador precise recriá-los a partir dos índices originais usando operações de divisão/módulo, além de impedir a realização de otimizações adicionais de código pelo compilador.

```
    #pragma omp parallel for collapse(2) /* Uso não otimizado */
      for (i = 0; i < imax; i++) {
           for (j = 0; j < jmax; j++)
                a[j + jmax*i] = 1.;
      }

    #pragma omp parallel for collapse(2) /* Uso recomendado */
      for (i = 0; i < imax; i++) {
           for (j = 0; j < jmax; j++)
                a[k++] = 1.;
      }
```

6.4.4 Cláusulas da diretiva sections

As cláusulas utilizadas com a diretiva **sections** são seguintes:

- **private(lista-de-variáveis)**
- **firstprivate(lista-de-variáveis)**
- **lastprivate(lista-de-variáveis)**
- **reduction(operador:(lista-de-variáveis))**
- **nowait**

A sintaxe e o resultado são como já vistos anteriormente com a diretiva **parallel**. Apenas uma única cláusula **nowait** pode aparecer na diretiva **sections**. A cláusula **nowait** remove a barreira implícita ao final da região paralela, como veremos em detalhes a seguir na Seção 6.5.3.

6.5 Sincronização

A sincronização é necessária para evitar condições de corrida no acesso às variáveis compartilhadas, sendo utilizada para assegurar uma ordem correta para as leituras e escritas, protegendo a atualização das variáveis compartilhadas (que não são atômicas por padrão).

Suponhamos que temos duas *threads* tentando incrementar a mesma variável *X*. A *thread* 1 deve escrever na variável *X* antes que *thread* 2 faça a sua leitura, ou a *thread* 1 deve ler a variável *X* depois que a *thread* 2 fizer sua escrita. É importante observar que as operações para atualização de variáveis compartilhadas (por exemplo, X = X + 1) não são atômicas! Se duas *threads* tentarem fazer isso ao mesmo tempo, uma das atualizações pode ser perdida e resultar em um valor final da variável incorreto.

Apresentamos a seguir os mecanismos, na forma de diretivas e funções, que o OpenMP dispõe para assegurar a correta sincronização entre as diversas *threads* em execução.

6.5.1 Diretiva barrier

Quando executamos um programa com múltiplas *threads* frequentemente é necessário coordenar a operação entre essas *threads*, por exemplo, determinando um ponto da computação onde todas as *threads* em execução possam aguardar pelas demais. Com isso, pode-se garantir que certas fases da computação terminaram, antes de avançar na execução do programa. Essa operação, chamada de barreira, é oferecida no OpenMP pela diretiva **barrier**.

Enquanto todas as *threads* não chegarem a uma barreira, nenhuma delas pode prosseguir além desse ponto. Devemos notar que há uma barreira implícita no final das diretivas **for, sections e single**. Adicionalmente, no final de uma região paralela, o time de *threads* é dissolvido e apenas a *thread master* (*thread 0*) continua, já que também há uma barreira implícita no final da região paralela. A sintaxe da diretiva **barrier** é a seguinte:

#pragma omp barrier

Um detalhe importante: ou todas, ou nenhuma das *threads*, devem encontrar a barreira, senão ocorrerá um **deadlock**. No programa ilustrado no exemplo seguinte, nenhuma mensagem será impressa por qualquer *thread* enquanto a *thread* 0 não fizer a leitura do teclado:

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])   /*omp_barrier.c */
{
```

```
#pragma omp parallel num_threads(4)
{
    int tid = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    if (tid == 0 ) {
        printf ("Estou atrasada para a barreira! Tecle enter\n");
        getchar();
    }
#pragma omp barrier
    printf("Passei da barreira. Eu sou a %d de %d threads \n", tid, num_threads);
}
    return 0;
}
```

A saída da execução desse programa seria;

```
Estou atrasado para a barreira! Tecle enter

Passei da barreira. Eu sou o 2 de 4 processos
Passei da barreira. Eu sou o 0 de 4 processos
Passei da barreira. Eu sou o 1 de 4 processos
Passei da barreira. Eu sou o 3 de 4 processos
```

No exemplo a seguir vemos o uso da diretiva **barrier** para assegurar que a leitura de uma variável compartilhada (no caso o vetor "a") seja feita apenas depois que for atualizada por todas as *threads*. Nessa situação, a barreira é necessária para assegurar a escrita de todo o vetor "a", antes que ele seja lido por qualquer *thread*.

```
    #pragma omp parallel private(i, tid, vizinho)
    {
        tid = omp_get_thread_num();
        vizinho = tid - 1;
        if (tid == 0)
            vizinho = omp_get_num_threads() - 1;
        ...
        a[tid] = a[tid]*3.5;
    #pragma omp barrier
        b[tid] = a[vizinho] + c
      ...
    }
```

Uma observação final: a diretiva **barrier** pode ser utilizada imediatamente a seguir de uma sentença **if** apenas se estiver em um bloco de código (entre chaves):

```
    void programa_exemplo(){
    int x = 1;
    ...
    if (x!=0) {
       #pragma omp barrier    /* uso válido   */
    }
    ...
    if (x!=0)
       #pragma omp barrier    /* uso inválido */
```

6.5.2 Diretiva master

Um bloco estruturado com a diretiva **master** deve ser executado apenas pela *thread master* (*thread* 0). As demais *threads* pulam o bloco estruturado e continuam a sua execução. Nesse

aspecto, seu comportamento é diferente da diretiva **single**, já que não existe uma barreira implícita ao final da região paralela. Na maior parte das vezes é utilizada para operações de E/S.

#pragma omp master
{
bloco estruturado
}

Essa diretiva deve estar em uma região paralela especificada por uma diretiva **parallel** e não pode haver desvio para dentro ou para fora dela. A diretiva **master** não admite nenhuma cláusula. No exemplo a seguir, uma diretiva **barrier** foi utilizada entre os laços para garantir a sincronização das *threads*.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) { /* omp_master.c  */
int a[5], i;
    #pragma omp parallel
    {
        // Realiza alguma computação
        #pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;
        // Imprime resultados intermediários
        #pragma omp master
            for (i = 0; i < 5; i++)
                printf("a[%d] = %d\n", i, a[i]);
        // Espera
        #pragma omp barrier
        // Continua a computação
        #pragma omp for
        for (i = 0; i < 5; i++)
             a[i] += i;
    }
}
```

```
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

6.5.3 Cláusula nowait

A cláusula **nowait** pode suprimir as barreiras implícitas no final das diretivas **for, sections e single**. O objetivo é aumentar o paralelismo possível em um programa, pois as barreiras podem ser muito caras, pelo tempo que as *threads* mais rápidas gastam esperando pelas demais.

Note que, por motivos quase óbvios, não é possível utilizar a cláusula **nowait** com as diretivas **parallel for** e **parallel sections**, já que no final de toda região paralela existe uma barreira implícita que não pode ser removida. A sintaxe da diretiva **nowait** é a seguinte:

#pragma omp for nowait
laço for

No exemplo a seguir apresentamos dois laços sem dependências, cujas barreiras implícitas podem ser removidas com o uso da cláusula **nowait**.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (j = 0; j < n; j++) {
        a[j] = c * b[j];
    }
    #pragma omp for nowait
    for (i = 0; i < m; i++) {
        x[i] = sqrt(y[i]) * 2.0;
    }
}
```

Use essa cláusula com atenção, porque é muito fácil remover uma barreira que é necessária. Isso resulta no pior tipo de erro, que é o comportamento não determinístico da aplicação: algumas vezes o resultado é correto e outras não; o comportamento se altera com uso do depurador etc. Vejamos o seguinte exemplo:

```
    #pragma omp for
    for (j = 0; j < k; j++) {
        a[j] = b[j] + c[j];
  }
    #pragma omp for
    for (j = 0; j < m; j++) {
        d[j] = e[j] * f;

    }
    #pragma omp for
    for (j = 0; j < n; j++) {
        z[j] = (a[j]+a[j+1]) * 0.5;
    }
```

No exemplo apresentado, pode-se remover a primeira barreira, ou a segunda, mas não ambas, por conta da dependência de dados no vetor "a" entre o primeiro e o último laço.

6.5.4 Diretiva critical

Uma seção crítica é um bloco de código executado apenas por uma *thread* de cada vez, sendo normalmente utilizado para proteger a atualização de variáveis compartilhadas. As seções críticas de uma diretiva **critical** podem receber nomes. Se uma *thread* está em uma seção crítica com um dado nome, nenhuma outra *thread* poderá estar em uma seção crítica com o mesmo nome (embora possa haver mais de uma *thread* em seções críticas com nomes diferentes).

#pragma omp critical [(nome)]
bloco estruturado

Se o nome é omitido, um nome nulo é assumido (todas as seções críticas sem nome têm efetivamente o mesmo nome). O exemplo mostrado na Seção 6.2.3 é aqui reapresentado com a diretiva **critical** para evitar a condição de corrida existente.

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[]) { /* omp_critical.c */
int conta_secao = 0;
#pragma omp parallel sections
{
    #pragma omp section
    {
```

```
        #pragma omp critical
        {
        conta_secao++;
        /* deve imprimir o número um ou dois */
        printf( "Ordem de execução da sessão %d\n", conta_secao);
        }
    }
    #pragma omp section
    {
        #pragma omp critical
        {
        conta_secao++;
        /* deve imprimir o número um ou dois */
        printf( "Ordem de execução da sessão %d\n", conta_secao );
        }
    }
}
    return(0);
}
```

O resultado agora de sua execução será sempre correto:

```
Ordem de execução da sessão 1 tid= 0
Ordem de execução da sessão 2 tid= 1
```

6.5.5 Diretiva atomic

A diretiva **atomic** permite acessar uma variável de modo atômico. As condições de corrida são evitadas através do controle direto das *threads* concorrentes que podem ler ou escrever em uma mesma variável. Com o uso da diretiva **atomic**, programas paralelos mais eficientes podem ser escritos e com um número menor de *locks*. Objetos que possam ser atualizados em paralelo e estar sujeitos a condições de corrida devem ser protegidos com a diretiva **atomic**.

#pragma omp atomic [cláusulas]
sentença ou bloco estruturado

Onde *sentença* é uma expressão com um tipo escalar. As cláusulas admitidas são as seguintes:

- **update** - atualiza o valor da variável atomicamente. Seu uso garante que apenas uma *thread* por vez atualizará a variável compartilhada, evitando erros de escritas simultâneas à mesma variável. É a operação padrão para a diretiva.
- **read** - lê o valor da variável atomicamente. O valor de uma variável compartilhada pode ser lido com segurança, evitando o perigo de uma leitura de um valor intermediário, caso a variável esteja sendo atualizada por outra *thread*.
- **write** - escreve o valor da variável atomicamente. O valor de uma variável compartilhada pode ser escrito exclusivamente para evitar erros de escritas simultâneas.
- **capture** - garante uma atualização atômica da variável com o operador especificado enquanto captura o valor original ou final da variável atomicamente.

Embora muito útil, a diretiva **atomic** não permite o seu uso com qualquer tipo de operação. A sentença ou bloco estruturado pode ter apenas uma das formas apresentadas na Tabela 6.2, de acordo com o tipo de cláusula utilizada na diretiva **atomic**.

Tabela 6.2: Operadores da diretiva atomic

cláusula	sentença	bloco estruturado
update	x++;	
	x–;	
	++x;	
	–x;	
	x binop = expr;	
	x = x binop expr;	
	x = expr binop x;	
read	v = x;	
write	x = expr;	
capture	v = x++;	{v = x; x binop = expr;}
	v = x–;	{v = x; xOP;}
	v = ++x;	{v = x; OPx;}
	v = –x;	{x binop = expr; v = x;}
	v = x binop = expr;	{xOP; v = x;}
	v = x = x binop expr;	{OPx; v = x;}
	v = x = expr binop x;	{v = x; x = x binop expr;}
		{x = x binop expr; v = x;}
		{v = x; x = expr binop x;}
		{x = expr binop x; v = x;}
		{v = x; x = expr;}

Nessa tabela temos as seguintes notações:

- **x, v** são ambas expressões *lvalue*, ou seja, um valor cujo endereço pode ser obtido diretamente (através do operador &), com tipos escalares;
- **expr** é uma expressão do tipo escalar que não faz referência a **x**;
- **binop** é um dos seguintes operadores binários: **+, *, -, /, &, ^,** Math input error e Math input error;
- **OP** pode ser um dos dois: **++** ou **–**.

A seguir, apresentamos vários detalhes que precisam ser observados para a correta utilização da diretiva **atomic**:

- Todos os acessos atômicos às posições de memória referenciadas por **"x"** em todo o programa devem ter um tipo compatível;
- Dentro de uma região atômica, ocorrências sintáticas múltiplas de **"x"** devem referenciar a mesma posição de memória;
- Todos os acessos a uma certa posição de memória durante um programa paralelo devem ser atômicos. Um acesso não atômico a uma posição de memória pode quebrar o comportamento atômico esperado de todos os acessos atômicos àquela posição de memória;
- Nem **"v"** ou **expr** podem acessar a posição de memória que é referenciada por **"x"**.

- Nem **"x"** ou **"expr"** podem acessar a posição de memória que é referenciada por **"v"**;
- Todos os acessos à posição de memória referenciada por **"x"** são atômicos. As avaliações das expressões **"expr"**, **"v"**, **"x"** não são atômicas;
- Para os acessos de **atomic capture**, a operação de escrita do valor capturado para a posição de memória representada por **"v"** não é atômica.

```
extern float x[], *p = x, y;
/* Proteção contra condições de corrida entre diversas atualizações */
#pragma omp atomic
    x[index[i]] += y;
/* Proteção contra condições de corrida com atualizações através de x. (*p = x) */
#pragma omp atomic
    p[i] -= 1.0f;

extern int x[10];
extern int f(int);
int temp[10], i;

for (i = 0; i < 10; i++) {
     #pragma omp atomic read
     temp[i] = x[f(i)];

     #pragma omp atomic write
     x[i] = temp[i]*2;

     #pragma omp atomic update
     x[i] *= 2;
}

extern int x[10];
extern int f(int);
int temp[10], i;

for (i = 0; i < 10; i++) {
     #pragma omp atomic capture
     temp[i] = x[f(i)]++;

     #pragma omp atomic capture
     {
       temp[i] = x[f(i)]; // as duas ocorrências de x[f(i)]
                          // devem ser avaliadas para a
       x[f(i)] -= 3;      // mesma posição de memória, de outro
                          // modo o comportamento é indefinido.
     }
}
```

O uso da diretiva **atomic** pode ser mais eficiente do que usar a diretiva **critical**, por exemplo, se diferentes elementos do vetor ou matriz forem protegidos separadamente.

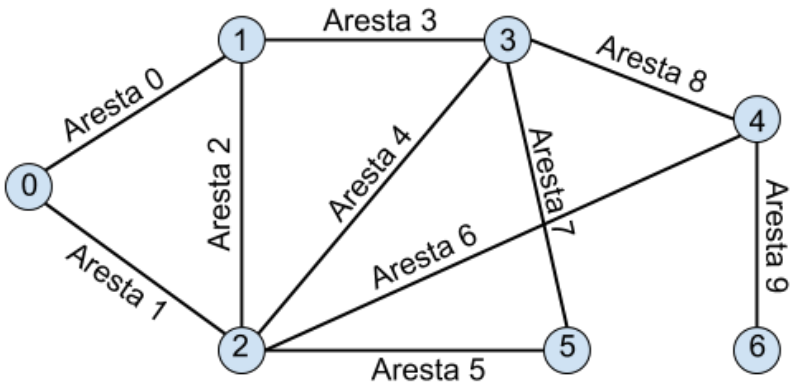

Figura 6.6: Graus dos vértices de um grafo

O exemplo listado a seguir calcula em paralelo os graus dos vértices de um grafo e faz uso da diretiva **atomic** para impedir que duas *threads* diferentes atualizem ao mesmo tempo o grau de um vértice, já que as iterações são divididas entre as *threads* de acordo como o número das arestas do grafo. Sem o uso da diretiva **atomic**, o resultado final poderia ser calculado de forma incorreta, pois existe uma condição de corrida na atualização dos elementos do vetor *grau[]*.

```
#include <stdio.h>
#include <omp.h>
#define ARESTAS 10
#define VERTICES 7
int main(int argc, char *argv[]) { /* omp_atomic.c  */
int i,j;
int num_arestas = ARESTAS, grau[VERTICES]={0,0,0,0,0,0,0};
typedef struct {
   int vertice1, vertice2;
   } tipo_aresta;
tipo_aresta aresta[ARESTAS];
    /* Atribui os valores iniciais aos vértices das arestas do grafo */
    aresta[0].vertice1 = 0;
    aresta[0].vertice2 = 1;
    aresta[1].vertice1 = 0;
    aresta[1].vertice2 = 2;
    aresta[2].vertice1 = 1;
    aresta[2].vertice2 = 2;
    aresta[3].vertice1 = 1;
    aresta[3].vertice2 = 3;
    aresta[4].vertice1 = 2;
    aresta[4].vertice2 = 3;
    aresta[5].vertice1 = 2;
    aresta[5].vertice2 = 5;
    aresta[6].vertice1 = 2;
    aresta[6].vertice2 = 4;
    aresta[7].vertice1 = 3;
    aresta[7].vertice2 = 5;
    aresta[8].vertice1 = 3;
    aresta[8].vertice2 = 4;
    aresta[9].vertice1 = 4;
    aresta[9].vertice2 = 6;
    /* Define o número total de threads */
    omp_set_num_threads(4);
    /* Executa em paralelo dividindo as iterações entre as threads */
```

```
    #pragma omp parallel for
    for (j = 0; j< num_arestas; j++){
        #pragma omp atomic
        /* A operação de incremento é realizada atomicamente */
        grau[aresta[j].vertice1]++;
        #pragma omp atomic
        /* A operação de incremento é realizada atomicamente */
        grau[aresta[j].vertice2]++;
    }
    /* Imprime o resultado */
    for (i = 0; i < VERTICES; i++)
        printf("Grau do vértice %d = %d \n",i, grau[i]);
    return(0);
}
```

Apresentamos em seguida um possível resultado para a execução desse programa.

```
Grau do vértice 0 = 2
Grau do vértice 1 = 3
Grau do vértice 2 = 5
Grau do vértice 3 = 4
Grau do vértice 4 = 3
Grau do vértice 5 = 2
Grau do vértice 6 = 1
```

6.5.6 Funções lock

Eventualmente pode ser desejável mais flexibilidade do que aquela fornecida pelas diretivas **critical e atomic**. Por exemplo, para o uso em qualquer ponto do código, incluindo em subrotinas diferentes, fora de blocos estruturados. Para isso, essas funções recebem como argumento o endereço de uma variável de *lock*.

Uma chave *lock* é uma variável especial que pode ser adquirida por uma *thread* e nenhuma outra *thread* pode adquirir uma chave *lock* até que a *thread* que a obteve libere essa chave. Algumas rotinas de aquisição de chave *lock* são bloqueantes e outras não, sendo que uma chave *lock* deve ter um valor inicial antes de ser usada e pode ser destruída quando não for mais necessária. Note que as variáveis usadas como chave *lock* não devem ser usadas para qualquer outro propósito.

A sintaxe das rotinas de **lock** do OpenMP são descritas a seguir. Em todas as rotinas a chave *lock* deve ser uma variável do tipo *omp_lock_t* e ter sido iniciada com a função **omp_init_lock**. A rotina **omp_init_lock** inicia a chave associada com o parâmetro *lock* para uso em chamadas subsequentes. A sua sintaxe é apresentada a seguir.

void omp_init_lock(omp_lock_t *lock);

A rotina **omp_set_lock** bloqueia a *thread* em execução até que a chave *lock* especificada esteja disponível e possa ser adquirida.

void omp_set_lock(omp_lock_t *lock);

A rotina **omp_test_lock** tenta adquirir uma chave *lock*, mas não bloqueia a execução da *thread*. Vemos a sua sintaxe seguir.

int omp_test_lock(omp_lock_t *lock);

A rotina **omp_unset_lock** libera a chave associada com o parâmetro *lock*. A sua sintaxe é apresentada a seguir.

void omp_unset_lock(omp_lock_t *lock);

A rotina **omp_destroy_lock** assegura que a chave especificada pela variável *lock* foi destruída e está no estado não iniciada. A sua sintaxe é detalhada a seguir.

void omp_destroy_lock(omp_lock_t *lock);

Existem também rotinas de **lock** aninháveis que permitem a uma mesma *thread* adquirir uma chave **lock** múltiplas vezes antes de liberá-la o mesmo número de vezes, mas não serão abordadas neste texto. Veja no código a seguir uma possível aplicação dessas funções.

```
#include <stdio.h>
#include <omp.h>

int main(int argc, const char * argv[]){
omp_lock_t chave_simples;
    omp_init_lock(&chave_simples);

    #pragma omp parallel num_threads(4)
    {

        while (!omp_test_lock(&chave_simples)){
            printf("=== Tid  %d: chave ocupada\n", omp_get_thread_num());
        }
        printf("+++ Tid %d: Consegui a chave\n", omp_get_thread_num());
        printf("--- Tid %d: Liberando a chave\n", omp_get_thread_num());
        omp_unset_lock(&chave_simples);

    }

    omp_destroy_lock(&chave_simples);
}
```

Uma possível saída para a execução desse programa seria:

```
=== Tid  1: chave ocupada
=== Tid  1: chave ocupada
+++ Tid 0: Consegui a chave
--- Tid 0: Liberando a chave
=== Tid  1: chave ocupada
+++ Tid 1: Consegui a chave
--- Tid 1: Liberando a chave
=== Tid  3: chave ocupada
+++ Tid 3: Consegui a chave
--- Tid 3: Liberando a chave
=== Tid  2: chave ocupada
+++ Tid 2: Consegui a chave
--- Tid 2: Liberando a chave
```

6.6 Variáveis de ambiente

Há algumas variáveis de ambiente importantes para a execução dos programas em OpenMP. Algumas já foram vistas anteriormente, tais como OMP_NUM_THREADS (veja Seção 6.3.1) e OMP_SCHEDULE (veja Seção 6.4.3). A seguir apresentamos outras variáveis de ambiente,

utilizadas para melhorar o desempenho do seu programa, caso o seu ambiente tenha um valor padrão diferente.

OMP_STACKSIZE

Caso grandes estruturas de dados privadas sejam utilizadas, é possível que o programa estoure o limite de espaço de pilha definido para sua utilização. Nesse caso, o tamanho das pilhas do time de *threads*, além da *thread master*, pode ser controlado pela variável de ambiente OMP_STACKSIZE. O tamanho da pilha específico da *thread master* continua sendo definido como em um programa sequencial, com o uso das opções do compilador ou com o comando *ulimit*.

O valor deve ser um inteiro Math input error concatenado com B, K, M, e G para especificar, respectivamente, bytes, kilobytes (1024 bytes), megabytes (1024 kilobytes), ou gigabytes (1024 megabytes), para especificar o tamanho da pilha:

```
$ export OMP_STACKSIZE=2000500B
$ export OMP_STACKSIZE="3000 k"
$ export OMP_STACKSIZE=10M
$ export OMP_STACKSIZE="10 M"
$ export OMP_STACKSIZE="20 M"
$ export OMP_STACKSIZE="1G"
$ export OMP_STACKSIZE=20000
```

OMP_DYNAMIC

Se o valor dessa variável de ambiente for definido como *true*, o ambiente de execução pode ajustar o número de *threads* para a execução de regiões paralelas de modo a otimizar o uso de recursos do sistema. Se o valor da variável for definido como *false*, isso impede o ambiente de execução entregue um número de *threads* menor que o solicitado. O valor padrão dessa variável de ambiente é dependente de implementação, assim como o comportamento do programa, se nenhum valor for definido.

```
$ export OMP_DYNAMIC=true
```

OMP_PLACES

A variável de ambiente OMP_PLACES especifica uma lista de lugares para a execução das *threads* no OpenMP. Essa localização das *threads* pode ser especificada usando nomes abstratos ou uma lista explícita de lugares.

Os nomes abstratos são *threads, cores e sockets*, e podem opcionalmente ser seguidos de um número em parênteses, expressando a quantidade de lugares a serem criados.

```
$ export OMP_PLACES="threads"
$ export OMP_PLACES="cores"
$ export OMP_PLACES="sockets"
```

Alternativamente, a localização pode ser expressa com lista de lugares separados por vírgula. Um lugar é especificado como um conjunto de números não negativos entre chaves, separados por vírgula, significando as *threads de hardware*. Para especificar um intervalo, use "**:**" seguido de uma contagem indicando o comprimento. Opcionalmente, o comprimento pode ser seguido por mais um "**:**" e um número indicando o salto. Se nenhum valor for definido, é assumido "1" como padrão para o salto. Por exemplo:

```
$ export OMP_PLACES="{0,1,2},{3,4,6},{7,8,9},{10,11,12}"
$ export OMP_PLACES="{0:3},{3:3},{7:3},{10:3}"
$ export OMP_PLACES="{0:2}:4:3"
```

OMP_PROC_BIND

Os valores que essa variável de ambiente pode ter são *true*, *false* ou uma lista separada por vírgulas dos seguintes valores: *master, close ou spread*.

Se a variável de ambiente for definida como *false*, o ambiente de execução pode mover *threads* OpenMP entre lugares OpenMP, a afinidade de *thread* é desabilitada e as cláusulas **proc_bind** em construções paralelas são ignoradas.

Em caso contrário, o ambiente de execução não pode mover as *threads* entre lugares OpenMP, a afinidade de *threads* está ativada e a *thread master* é vinculada ao primeiro lugar na lista de lugares OpenMP (como definido pela variável de ambiente OMP_PLACES).

Com o uso do valor *master*, o time de *threads* de trabalho é alocado no mesmo lugar OpenMP no qual a *thread master* está executando. Com uso de *close*, as *threads* são alocadas em lugares próximos da *thread master*. E com *spread*, uma distribuição mais espalhada possível entre os diversos lugares OpenMP disponíveis é usada.

O comportamento do programa é dependente de implementação, se o valor na variável de ambiente OMP_PROC_BIND não for definido corretamente. O comportamento também é dependente de implementação se a *thread master* não puder ser vinculada ao primeiro lugar definido na lista de lugares OpenMP. Exemplos de uso são:

```
$ export OMP_PROC_BIND=false
$ export OMP_PROC_BIND="close"
$ export OMP_PROC_BIND=spread
```

Dependendo do tipo de aplicação, uma distribuição pode oferecer maiores ganhos de desempenho do que a outra, devendo ser investigado caso a caso.[1]

OMP_WAIT_POLICY

O valor padrão (*active*) faz com que as *threads* ociosas continuem executando, ao invés de serem colocadas para dormir (*busy-wait*). Apesar de definido pelo usuário, o valor dessa variável pode ser ignorado pelo ambiente de execução. O valor padrão desta variável de ambiente é dependente da implementação. O outro valor possível para essa variável é *passive* (qualquer outro valor é dependente da implementação). Exemplos:

```
$ export OMP_WAIT_POLICY=ACTIVE
$ export OMP_WAIT_POLICY=active
$ export OMP_WAIT_POLICY=PASSIVE
$ export OMP_WAIT_POLICY=passive
```

OMP_THREAD_LIMIT

Essa variável de ambiente define o número máximo de *threads* OpenMP que podem ser utilizadas. O valor dessa variável deve ser um inteiro positivo. O comportamento do programa será indefinido caso o número de *threads* seja maior do que o máximo suportado pela implementação ou se o valor não for um inteiro positivo.

```
$ export OMP_THREAD_LIMIT=32
```

6.7 Erros comuns e recomendações

A seguir listamos alguns erros comuns cometidos pelos programadores menos experientes e elencamos algumas recomendações quando se programa com OpenMP.

Algumas omissões frequentes

- Esquecer o uso da diretiva **parallel**, isso vai resultar em uma execução sequencial do trecho de código:

```
// sem diretiva "parallel"
{
    #pragma omp for
    for (size_t i = 0; i < n; ++i ) {
        ...
    }
}
```

- Outro esquecimento frequente é omitir a palavra **for**, isso vai ocasionar a execução do laço por todas as *threads*:

```
#pragma omp parallel
{
    #pragma omp // sem diretiva "for"
    for (size_t i = 0; i < n; ++i ) {
        ...
    }
}
```

```
Infelizmente não haverá avisos do compilador sobre esse tipo de erro
e seu programa poderá até ser executado corretamente, mas não haverá
nenhum ganho de desempenho.
```

- Uma possível maneira de se evitar o problema relatado anteriormente é especificar sempre o tipo de escalonamento desejado quanto utilizar a construção **for**. Isso oferece também uma melhor portabilidade de código, porque o escalonamento padrão é dependente de implementação e nem sempre é o "static". Caso a diretiva **for** seja esquecida, o compilador vai detectar imediatamente o erro, porque **#pragma omp parallel schedule(static)** não é uma construção válida em OpenMP.

- Um erro comum para os iniciantes é esquecer de declarar algumas variáveis como privadas, embora elas sejam utilizadas desse modo. O valor padrão de compartilhamento para as variáveis é **shared**. Isso pode ser contornado declarando-se as variáveis dentro da região paralela, o que vai torná-las privadas como padrão. Outra maneira é utilizar a cláusula **default(none)**. Isso vai obrigar a declaração do escopo de todas as variáveis no início da região paralela e forçar o compilador a gerar uma mensagem de erro, caso isso não seja feito.

Observações sobre laços for

- Uma forma simples de verificar se um laço é paralelizável ou não, é executar o laço na ordem inversa e verificar se o resultado é o mesmo. Se for o caso, então ele é quase certamente paralelizável.

- Apesar de a especificação do OpenMP permitir o tipo de escalonamento "auto", ele não tem um comportamento definido e, portanto, é dependente de implementação. Assim, recomendamos novamente o uso do tipo "static": **#pragma omp parallel for schedule(static)**.

- Note que não é possível mudar o valor de uma variável de índice de laço dentro de um **#pragma omp for**. Isso é explicitamente proibido na especificação do padrão OpenMP.

- Uma outra restrição relativa à diretiva **#pragma omp for** é quanto ao formato do laço *for* a ser paralelizado: ele deve estar no formato canônico, isto é, deve ser do tipo **for (*init*; *test*; *inc*)**, onde **init** deve ser a atribuição de valor inicial a uma única variável de controle do laço, **test** deve ser o teste de uma condição de parada utilizando apenas os operadores relacionais (<, <=, >, >=, !=), e **inc** deve ser o incremento/decremento da variável de controle; como podemos ver nos a seguir:

```
#pragma omp for
    for (i = N-1; i >= 0; i--) {
    ....
    }
#pragma omp for
    for (i = a+x; i < N*x; i+=x) {
    ...
    }
```

- O uso da cláusula **nowait**, conforme o exemplo a seguir, pode ser seguro, desde que o valor de Math input error seja o mesmo para os dois laços e o escalonamento utilizado seja **static**. Neste caso as mesmas *threads* serão atribuídas às mesmas iterações.

```
#pragma omp parallel {
    #pragma omp for schedule(static) nowait
    for (i = 0; i < N; i++) {
        a[i] = ....
}
    #pragma omp for schedule(static)
    for (i = 0; i < N; i++) {
        ... = a[i]
    }
}
```

- Para cada laço sendo paralelizado, verifique se cada iteração do laço tem a mesma quantidade de trabalho. Caso isso não aconteça, a substituição do escalonamento estático (padrão) pelo escalonamento dinâmico ou guiado pode melhorar o desempenho do programa.

Uso da cláusula de redução

- Use a redução sempre que possível. A partir versão 4.5 a redução sobre vetores também é possível e evita o uso de diretivas de sincronização, com significativos ganhos de desempenho:

```
#define N 100
int main(){
int i,j;
float a[N], b[N][N];

        ...

    #pragma omp parallel for reduction(+:a[0:N]) private(j)
```

```
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            a[j] +=  b[i][j];
        }
    }
}
```

- O único uso permitido da variável de redução é sua própria atualização e com o mesmo operador especificado na cláusula **reduction**. Por exemplo, não faz sentido somar os valores de uma variável de redução se o resultado final será computado pela multiplicação dos valores parciais obtidos.

```
#include <omp.h>
int main(int argc, char *argv[]) {
int i, soma = 0;

    #pragma omp parallel for reduction(+:soma)
    for (i = 0; i < 1000; i++) {
        soma = 2 - soma;        // operador errado
    }
    return(0);
}
```

Observações sobre sincronização

- Se mais de uma *thread* acessar uma variável e uma delas realizar uma escrita, mesmo algo tão simples quanto Math input error, deve-se utilizar a sincronização, pois o OpenMP não dá garantia do resultado.

- Por uma questão de simplicidade, é recomendável o uso da cláusula **critical** em vez de *lock* sempre que possível. Nesse caso, tenha o cuidado de usar diferentes nomes para cada região crítica sempre que possível. Se for necessário o uso de *locks*, assegure-se de que as funções **omp_set_lock()** e **omp_unset_lock()** são chamadas pela mesma *thread*.

- Contudo, por uma questão de desempenho, o uso da diretiva **atomic** é preferível ao uso da diretiva **critical**, porque o compilador será capaz de realizar otimizações, o que não é tão fácil com uso da diretiva **critical**.

- Tente evitar os custos associados com a chamada repetidas vezes da região crítica, por exemplo, verificando uma variável de condição antes de entrar na região crítica. Tente colocar o mínimo de código possível dentro da região crítica quanto for possível. Funções complexas, por exemplo, devem ser calculadas antecipadamente.

- As atualizações para variáveis compartilhadas não podem ser protegidas em um ponto do código com a diretiva **atomic** e em outro com a diretiva **critical**, se estão disputando a mesma variável.

```
#pragma omp parallel
{
#pragma omp critical
    a+=2;
#pragma omp atomic
    a+=3;
}
```

- Certifique-se também se é realmente necessário proteger o acesso à memória, por exemplo, se a variável for privada; ou se a variável for compartilhada, mas apenas de leitura; ou ainda se apenas uma *thread* garantidamente faz o acesso à variável que se deseja proteger.

Medindo o tempo de execução

- Para medir o tempo de execução dos seus programas utilize sempre **omp_get_wtime()** (veja a Seção 6.3.2). Não use a rotina **clock()**, pois ela vai medir o tempo de processador acumulado em todas as *threads*. Assim não será observado *speedup* algum!

- Se a função **gettimeofday()** for utilizada para contabilizar o tempo total de execução, lembre-se que a hora do dia é recuperada com precisão de microssegundos a partir do fuso horário indicado. Por isso, qualquer atualização da hora do relógio (de um servidor de relógio global, ou por um administrador de sistema), interferirá na medição do tempo. Novamente, sugerimos a utilização da função **omp_get_wtime()** (veja a Seção 6.3.2).

Observações adicionais

- Quando forem realizadas operações de E/S, pode ser interessante o uso de um *buffer* para armazenar os dados (o que até pode ser feito em paralelo) e realizar a escrita de uma vez só para o dispositivo real.

- Eventualmente pode surgir alguma dúvida quanto ao uso das diretivas **single** ou **master**. Embora ambas as construções resultem na execução de um bloco de código por uma única *thread*, enquanto as demais pulam essa região, há uma diferença significativa entre elas. A diretiva **master**, de melhor desempenho, não tem nenhuma barreira implícita, ao contrário da diretiva **single**, que possui uma barreira implícita no final da região paralela. Assim, se for importante a sincronização entre as *threads*, use a diretiva **single**, em caso contrário, use a diretiva **master**.

6.8 Estudos de caso

6.8.1 Estudo de caso: método do trapézio

Em nosso primeiro estudo de caso, vamos revisar o programa de cálculo do valor da integral utilizando o método do trapézio, conforme mostrado na Seção 3.8. Vamos perceber que as alterações necessárias no código são muito pequenas e se resumem basicamente em uma única linha no código.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
double f(double x) { /* Calcula f(x) */
double valor;
    valor = exp(x);
    return(valor);
}
int main(int argc, char *argv[]) { /* omp_trapezio.c  */
double integral;            /* Armazena resultado em integral */
double a=0.0, b=1.0;        /* Limite esquerdo e direito */
long n=8000000000;          /* Número de Trapezóides */
double h;                   /* Largura da base do Trapezóide */
```

```
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    double t_inicio = omp_get_wtime();
    /* Faz a redução e escalona as iterações para as threads */
#pragma omp parallel for reduction(+:integral) schedule(static)
    for (long i = 1; i < n-1; i++) {
         integral += f(a + i*h);
    }
    integral *= h ;
    double t_fim = omp_get_wtime();
    printf("Com n = %ld trapezoides, a estimativa \n", n);
    printf("da integral de %f ate %f = %lf \n", a, b, integral);
    printf("Tempo: \t %f com %d threads.\n", t_fim-t_inicio, omp_get_max_threads());
    return(0);
}
```

Caso esse exemplo seja salvo com o nome *omp_trapezio.c*, a compilação e execução pode ser feita com os seguintes comandos em um terminal de um sistema operacional tipo Linux:

```
$ gcc -o omp_trapezio omp_trapezio.c -fopenmp -O3 -std=c11 -lm
$ ./omp_trapezio
```

Tal como no exemplo do trapézio da Seção 3.8, vamos calcular o valor aproximado da integral da função Math input error no intervalo entre 0 e 1, com Math input error. Esse valor tão grande de Math input error é necessário para garantir um tempo mínimo de execução, visando a análise de seu comportamento, mas também para garantir a precisão do método utilizado, sabendo-se que o resultado esperado deve ser aproximadamente Math input error.

O método do trapézio tem um grande potencial de paralelismo, pois cada iteração do laço de repetição **for** pode ser executada de forma independente. Dessa forma, podemos utilizar a diretiva **#pragma omp for** para paralelizar o cálculo da função **f()**.

Nesse exemplo, a atribuição de cada iteração do laço para as *threads* disponíveis foi feita utilizando a estratégia de escalonamento estática **schedule(static)** do OpenMP. Essa divisão atribui Math input error iterações para cada *thread*, onde Math input error é a quantidade de *threads* disponíveis na região paralela. Como essa divisão pode não ser exata, o ambiente de execução do OpenMP se encarrega de atribuir as iterações restantes entre as *threads*, conforme políticas dependentes de implementação.

A variável *integral* do exemplo é utilizada para acumular todos os resultados dos cálculos realizados em cada iteração. Como é uma variável compartilhada entre todas as *threads*, é necessário tratar a condição de corrida por meio de sincronização. Nesse caso, foi utilizada a cláusula **reduction(+:integral)** e todo esse controle de atualização e sincronização do acesso à variável foi delegado novamente para o OpenMP.

A função **omp_get_wtime()** (Seção 6.3.2) foi utilizada para medir o tempo de execução desse laço. O resultado da execução desse programa é o seguinte:

```
Com n = 128000000000 trapezoides, a estimativa
da integral de 0.000000 ate 1.000000 = 1.718282
Tempo:   94.971425 com 48 threads.
```

A seguir vamos explorar melhor este exemplo, fazendo a análise de sua escalabilidade.

Avaliação de desempenho

A análise de escalabilidade que faremos neste estudo está dividida em duas abordagens, uma sob a ótica de Amdahl e outra sob a de Gustafson, variando o número de *threads* (e processadores) entre 1 e 24 em ambos os casos. Primeiramente vamos variar apenas o número de *threads*, mas mantendo o tamanho do problema fixo. No segundo caso, vamos aumentar o tamanho do problema proporcionalmente, à medida que aumentamos o número de *threads*.

Os testes foram realizados em um computador com uma arquitetura de multiprocessamento simétrico com 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz).

Os tempos de execução foram coletados, e os cálculos de *speedup* e eficiência foram feitos conforme equações apresentadas na Seção 2.4.1, sendo que os valores medidos e as métricas calculadas são apresentados na Tabela 6.3. Neste experimento, o tempo de execução de 2324 segundos do algoritmo sequencial foi medido com uma *thread* e Math input error, sendo utilizado como referência para os demais cálculos dessa tabela.

Tabela 6.3: Métricas do método de trapézio - Amdahl

Número de *threads*	**Tempo de execução**	***Speedup***	**Eficiência**
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

A Figura 6.7 mostra a evolução do *speedup* de acordo com o aumento de processadores, de acordo com os dados obtidos a partir da Tabela 6.3. Uma linha contínua nesse gráfico indica o *speedup* linear ideal e a outra linha com pontos destacados apresenta o resultado do *speedup* calculado.

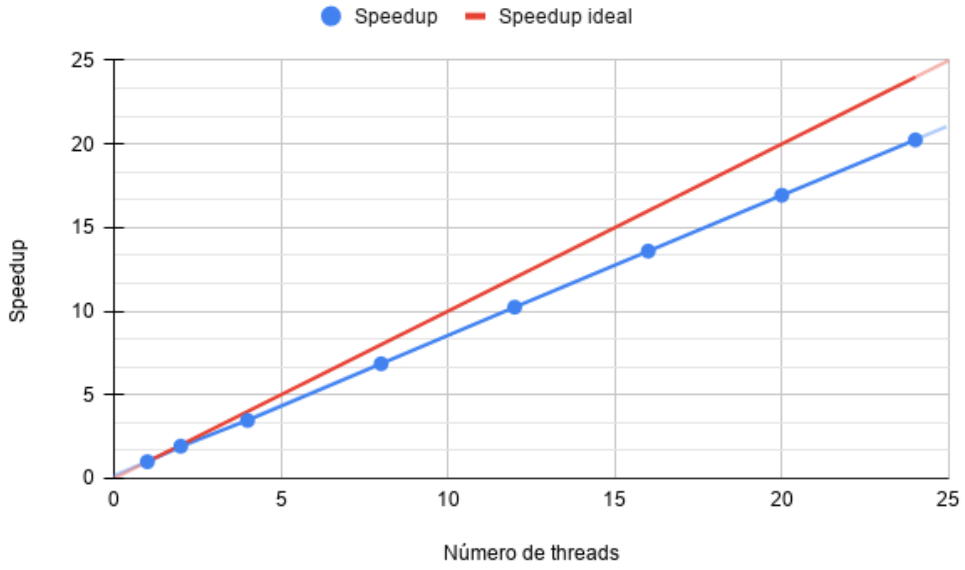

Figura 6.7: Gráfico de *speedup* - Amdahl

Esse gráfico mostra um comportamento esperado para o problema do trapézio, pois o cálculo de cada iteração pode ser realizado de forma independente. Apesar disso, sabemos que essa solução possui uma fração não paralelizável e, como consequência, esse trecho afeta a escalabilidade.

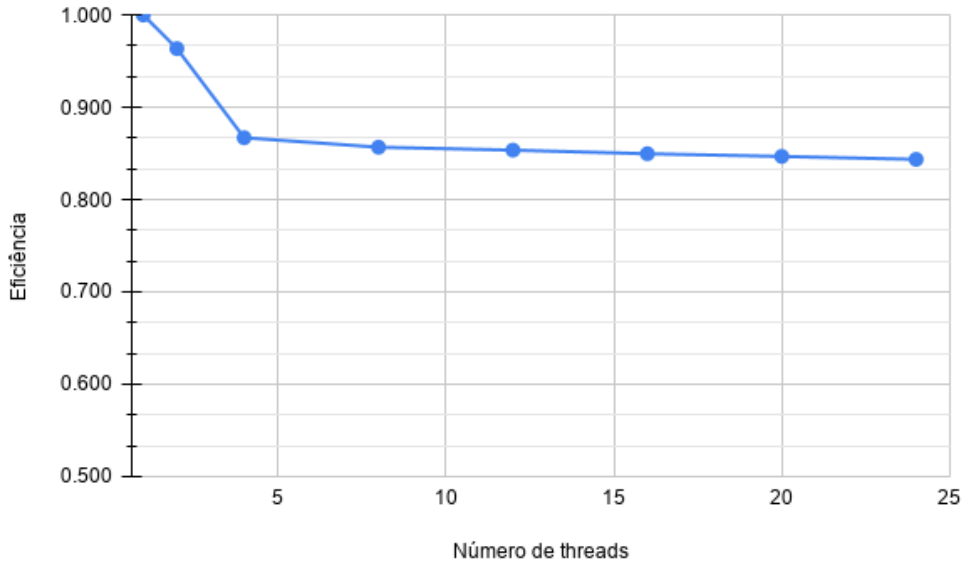

Figura 6.8: Gráfico de eficiência - Amdahl

Uma das partes importantes dessa solução é o somatório final do resultado de cada iteração feita pela cláusula de redução **reduction(+:integral)**. Essa e outras partes não paralelizáveis impactam na escalabilidade do programa.

Esse impacto é percebido também no gráfico de eficiência, como mostra a Figura 6.8. A queda da eficiência no gráfico ao adicionar 2 processadores, seguida de nova queda com 4 processadores, indica a existência de um trecho sequencial na solução que não se beneficia da disponibilidade de mais processadores.

Para medir a escalabilidade de acordo com a definição de Gustafson (veja a Seção 2.4.3), aumentamos o tamanho do problema na mesma proporção que o aumento do número de processadores. No caso do método do trapézio, o tamanho do problema está diretamente relacionado ao laço de repetição e, portanto, realizamos um aumento linear de Math input error para obter um aumento também linear na carga de trabalho.

Tabela 6.4: Métricas método do trapézio - Gustafson

Tamanho N	**Tempo Seq.**	**Threads (*p*)**	**Tempo Paralelo**	**Speedup**	**Efici- ência**
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Tamanho N	Tempo Seq.	Threads (*p*)	Tempo Paralelo	Speedup	Efici- ência
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Em nosso experimento definimos inicialmente que Math input error para um processador e continuamos aumentando o tamanho de problema em razão da quantidade de processadores utilizados, conforme mostra a Tabela 6.4. Esses resultados também podem ser observados no gráfico da Figura 6.9, onde a linha contínua indica o *speedup* linear ideal e a linha com pontos indica o resultado do *speedup* calculado.

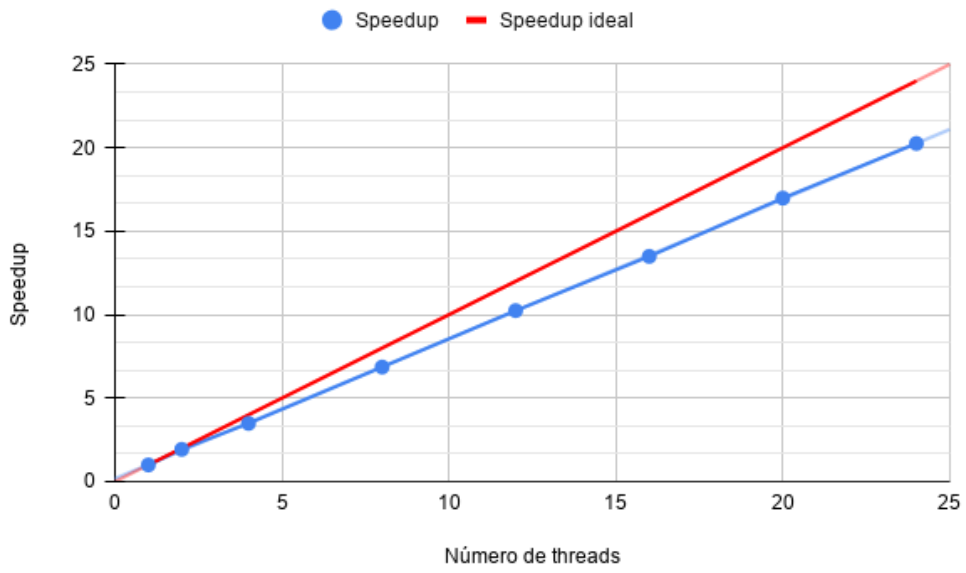

Figura 6.9: Gráfico de *speedup* - Gustafson

O comportamento da curva de *speedup* da Figura 6.9 se mostra muito similar à Figura 6.7. Isso indica que o tempo gasto com a parte sequencial da solução tem um impacto proporcional conforme o tamanho do problema aumenta. No exemplo apresentado, a cláusula de redução **reduction(+:integral)** pode ser o principal elemento para que a curva de *speedup* tenha esse comportamento.

Na Figura 6.10 apresentamos a curva de eficiência para esse novo experimento, bastante semelhante à da Figura 6.8. Mais uma vez, percebe-se que a eficiência da nossa solução diminui a partir de 2 processadores e até o limite inferior de 85% de eficiência.

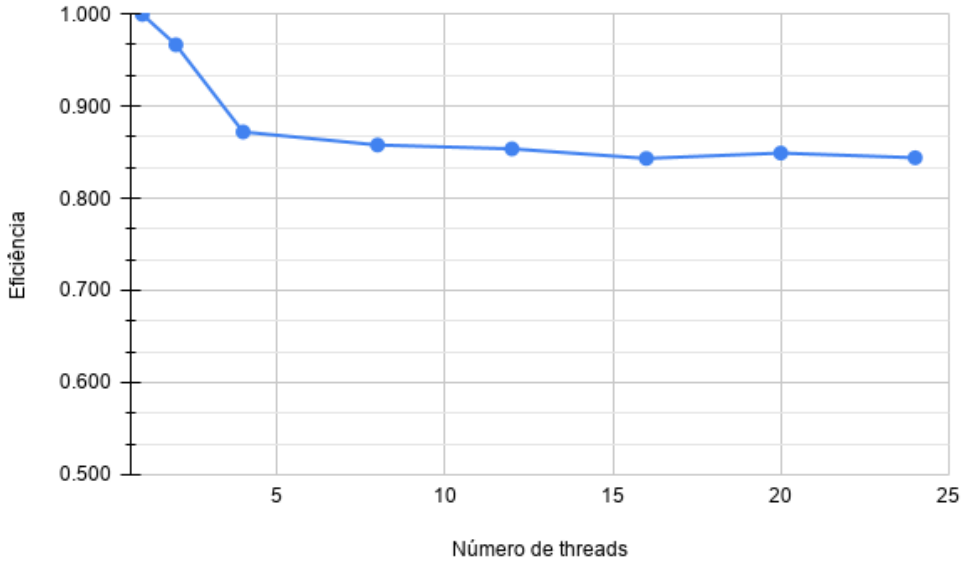

Figura 6.10: Gráfico de eficiência - Gustafson

Como no exemplo com uso do MPI (veja a Seção 3.8), os resultados obtidos em ambos métodos de avaliação, Amdahl e Gustafson, são muito similares, dado que a fração sequencial do problema é muito pequena. A queda inicial de desempenho, com duas e quatro *threads*, provavelmente se deve à sobrecarga para a criação e controle de *threads*, que é amortizada quando temos um número maior de *threads* em execução.

6.8.2 Estudo de caso: contagem de números primos

O outro estudo de caso é um programa para determinar a quantidade de números primos entre *0* e um determinado valor inteiro Math input error. Neste estudo, visitaremos a solução sequencial apresentada para o problema dos números primos na Seção 5.8.

Esta solução sequencial descarta todos os números pares de início, pois obviamente eles não são primos. Em seguida, é verificado se Math input error é divisível por algum número ímpar entre 0 e a raiz quadrada de Math input error.

Neste estudo de caso, vamos paralelizar o mesmo código-fonte, alterando apenas as estratégias de escalonamento. O principal objetivo da utilização do mesmo código é verificar o impacto no tempo de execução da estratégia de escalonamento escolhida para atribuição das iterações do laço às *threads* disponíveis.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
int primo (long int n) {
    long int i;

    for (i = 3; i < (long int)(sqrt(n) + 1); i+=2)
        if (n%i == 0)
            return 0;
    return 1;
```

```
}
int main(int argc, char *argv[]) { /* omp_primos.c  */
double t_inicio, t_fim;
long int i, n, total;
    if (argc < 2) {
        printf("Valor inválido! Entre com o valor do maior inteiro\n");
        return 0;
    } else {
        n = strtol(argv[1], (char **) NULL, 10);
    }
    total=0;
    t_inicio = omp_get_wtime();
     /* Solução trivial  */
#pragma omp parallel for reduction(+:total) schedule(static)
    /* Solução melhorada: schedule(dynamic,10000) */
    for (i = 3; i <= n; i += 2)
         if(primo(i) == 1) total++;
    /* Acrescenta o dois, que também é primo */
    total += 1;
    t_fim = omp_get_wtime();
    printf("Quant. de primos entre 1 e %ld: %ld\n", n, total);
             printf("Tempo   de   execução:   %f   com   %d   threads\n",   t_fim-t_inicio,
omp_get_max_threads());
    return(0);
}
```

Esse exemplo deve ser salvo com o nome *omp_primos.c*. A compilação e execução utilizando todas as *threads* disponíveis pode ser feita com os seguintes comandos em um terminal de um sistema operacional tipo Linux:

```
$ gcc -o omp_primos omp_primos.c -fopenmp -O3 -std=c11 -lm
$ ./omp_primos
```

Esse primeiro código apresenta uma solução trivial para a paralelização, pois distribui uma faixa contínua de números para cada *thread*, usando o escalonamento estático (**schedule(static)**). Já sabemos que, se essa divisão não for exata, o ambiente de execução do OpenMP se encarrega de atribuir as iterações restantes entre as *threads*.

Em seguida, cada *thread* conta quantos números primos existem no seu intervalo, incrementando a variável *total*, cujos respectivos valores são reduzidos para a variável de mesmo nome na *thread master* com uso da cláusula **reduction(+:total)**.

Essa solução é de simples implementação, contudo observamos um desbalanceamento de carga entre as *threads*. Esse desequilíbrio acontece porque a busca por números primos nas faixas mais altas tem custo computacional maior que nas faixas mais baixas. Ao executarmos a solução trivial com Math input error e 48 *threads* e medirmos o tempo de execução com a função **omp_get_wtime()** (Seção 6.3.2), obtemos o seguinte resultado:

```
Quant. de primos entre 1 e 700000000: 36252931
Tempo de execução: 67.173297 com 48 threads
```

Uma opção para diminuir esse problema do desequilíbrio de carga entre as *threads* seria utilizar o escalonamento dinâmico com a definição do tamanho do pedaço das iterações do laço de repetição, onde trocamos o escalonamento estático **schedule(static)** pelo dinâmico **schedule(dynamic, 10000)**, sendo a escolha do tamanho do pedaço (10.000) arbitrária. O ambiente OpenMP controla a lista com todos estes pedaços que, um a um, são atribuídos às *threads* disponíveis.

Quando uma *thread* termina sua busca por números primos, o ambiente de execução OpenMP lhe atribui um novo pedaço. Lembramos que a escolha do tamanho 10.000 foi arbitrária, mas serve para os propósitos de demonstração deste estudo de caso. Ao executarmos a solução balanceada com Math input error e 48 *threads* e medirmos o tempo de execução, temos:

```
Quant. de primos entre 1 e 700000000: 36252931
Tempo de execução: 52.203574 com 48 threads
```

De posse desses resultados, fica evidente a interferência do desbalanceamento de carga no tempo total das execuções: a solução trivial demorou 67 segundos, enquanto a segunda solução demorou 52 segundos (22% mais rápida).

Para investigar melhor os motivos dessa diferença, realizamos um novo experimento com Math input error e 8 *threads*. A Figura 6.11 mostra o tempo gasto em cada *thread* para procurar todos os números primos em seu intervalo de faixa contínua na solução trivial. Pode-se notar um grande desbalanceamento no tempo de execução das *threads*. Esse gráfico evidencia que as *threads* procurando números primos nas faixas mais baixas terminam mais rápido do que as que buscam nas faixas mais altas. Por exemplo, a *thread* 0 terminou a busca em 127 segundos, enquanto a *thread* 7 terminou a busca em 405 segundos. O tempo total de execução da solução trivial foi então de 405 segundos, que é o tempo de execução da *thread* mais demorada.

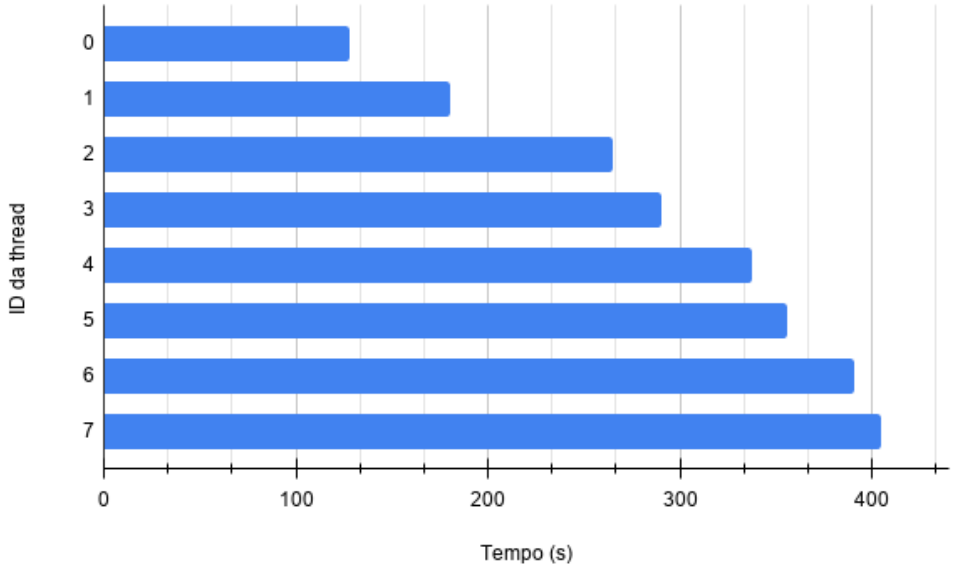

Figura 6.11: Tempo de execução de cada *thread* mostrando o desequilíbrio de carga entre elas

Após a execução do programa com a solução balanceada, observamos que as *threads* na Figura 6.12 têm aproximadamente o mesmo tempo de execução, por volta de 291 segundos. Consequentemente, o tempo total de execução da solução balanceada diminuiu para 291 segundos.

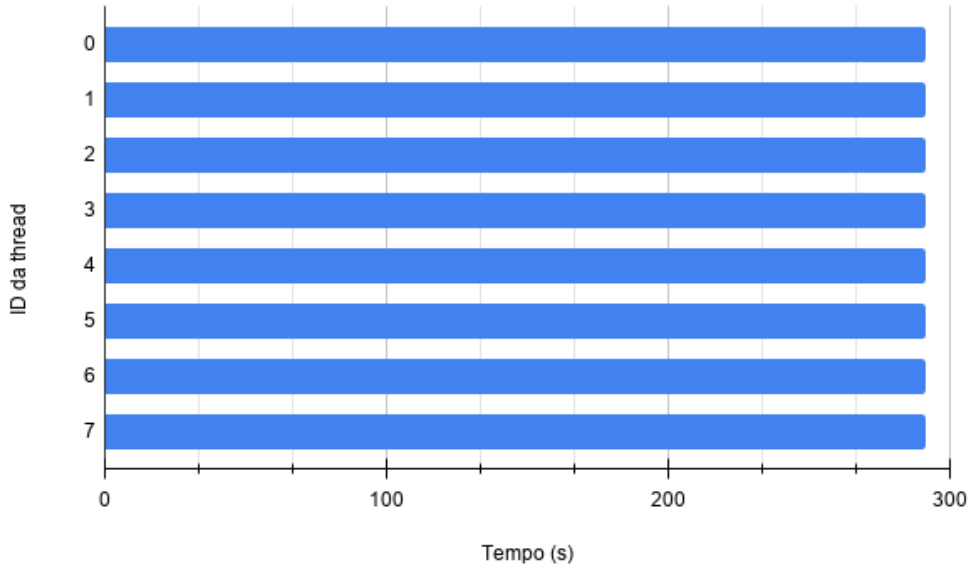

Figura 6.12: Tempo de execução de cada *thread* de forma balanceada

A solução balanceada é parecida com o método de "saco de tarefas" apresentado na Seção 5.8. A principal diferença reside no controle e atribuição das tarefas para as *threads*, o que é feito pelo próprio ambiente de execução do OpenMP, à medida que as *threads* se tornam disponíveis para execução. Essa solução balanceada foi obtida a partir de um breve estudo sobre o comportamento das *threads* e uma simples mudança na estratégia de escalonamento. Esses pontos devem estar sempre em mente na busca de uma estratégia para otimizar o código OpenMP.

Avaliação de desempenho

Uma arquitetura de multiprocessamento simétrico com um total de 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz) foi escolhida para realizar o nosso estudo de escalabilidade.

As Tabelas 6.5 e 6.6 mostram os tempos de execução da solução trivial e da solução balanceada, respectivamente. Esses tempos foram utilizados nos cálculos das métricas, como *speedup* e eficiência, conforme discutido na Seção 2.4.1, apresentados nas mesmas tabelas.

O tempo de 1316 segundos com uma *thread* e Math input error foi utilizado como base para os demais cálculos dessas tabelas.

Tabela 6.5: Métricas números primos - solução trivial

Número de *threads*	**Tempo de execução**	***Speedup***	**Eficiência**
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

Número de *threads*	Tempo de execução	*Speedup*	Eficiência
Math input error	Math input error s	Math input error	Math input error

Tabela 6.6: Métricas números primos - solução balanceada

Número de *threads*	Tempo de execução	*Speedup*	Eficiência
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

Para entender o comportamento das soluções, desenhamos as curvas de escalabilidade na Figura 6.13. Nesse gráfico, a reta contínua representa o *speedup* linear ideal; a reta logo abaixo, anotada com pontos, indica o *speedup* da solução balanceada; a reta abaixo de todas apresenta o *speedup* da solução trivial, anotada com quadrados.

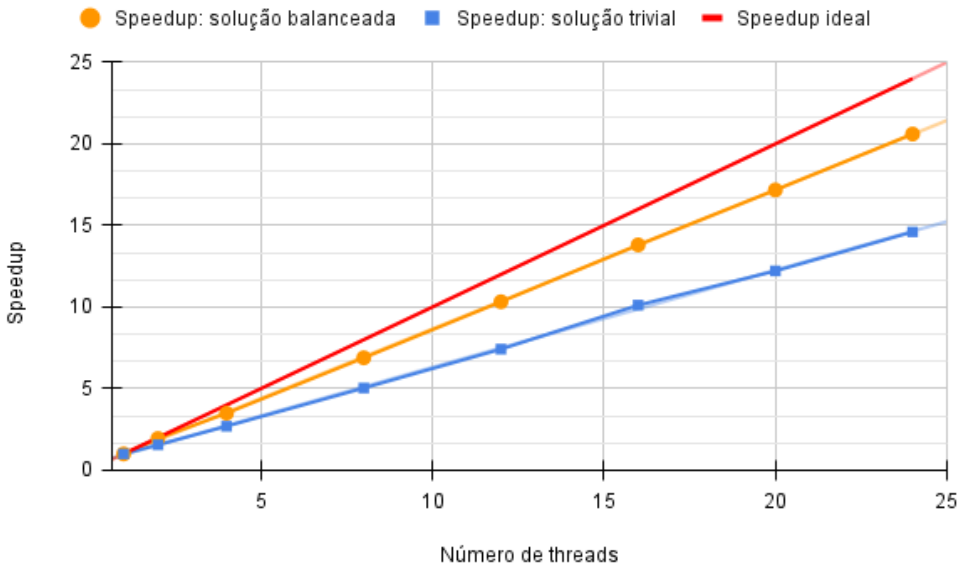

Figura 6.13: Gráfico de *speedup* - números primos

O gráfico das duas soluções tem, conforme o esperado, um crescimento linear. A escalabilidade da solução balanceada está muito mais próxima do *speedup* ideal. A escalabilidade da solução trivial se afasta da linha do *speedup* ideal conforme a quantidade de *threads* é aumentada, devido ao desbalanceamento, que se torna cada vez mais evidente.

Esse desbalanceamento de carga fica ainda mais evidente quando são comparadas as eficiências das duas soluções, conforme pode ser visto na Figura 6.14. A curva no gráfico com marcações em

círculos mostra a eficiência da solução balanceada e a curva com quadrados apresenta a eficiência da solução trivial.

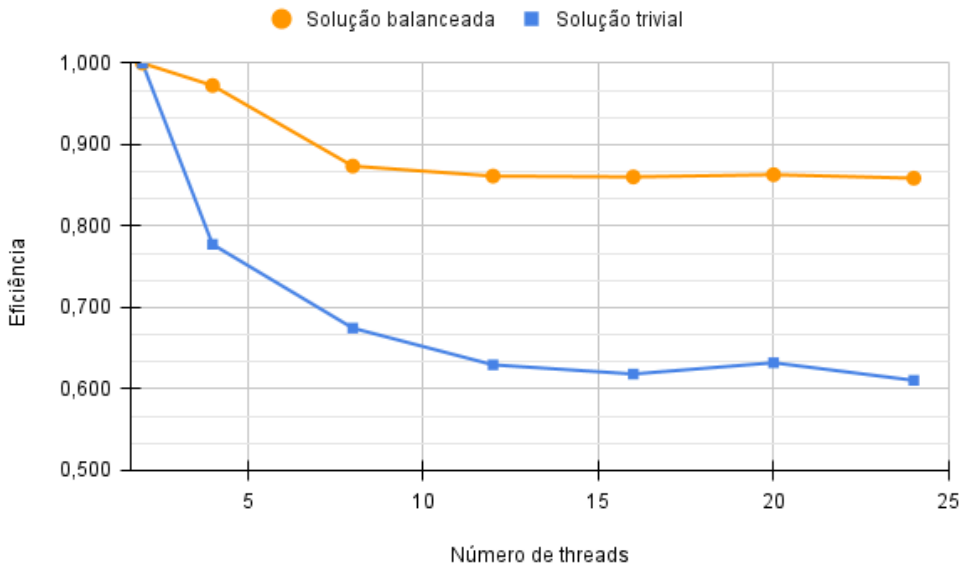

Figura 6.14: Gráfico de eficiência - números primos

A queda na eficiência da solução trivial acontece de forma rápida, mostrando o impacto do desbalanceamento. Nota-se que com 24 *threads* a eficiência é de 60% ou quase a metade da capacidade de processamento disponível.

A solução balanceada distribui a carga de trabalho de forma mais equilibrada entre todas as *threads* e, portanto, sua eficiência ficou por volta de 85%. Alguns trechos deste programa são executados de forma sequencial, como na operação de redução **reduction(+:total)**. Esses trechos afetam a eficiência, como observado a partir do uso de 2 *threads*.

6.8.3 Estudo de caso: método de Jacobi

O último estudo de caso do OpenMP é a solução da equação geral de calor. Dado que a função Math input error representa a temperatura Math input error no instante de tempo Math input error e na posição Math input error de uma placa, a equação de transferência de calor pode ser escrita como uma equação diferencial parcial, conforme a equação a seguir.

Math input error

Essa equação pode ser resolvida utilizando o método de diferenças finitas, que consiste na reescrita do problema contínuo original em um problema discreto, com fórmulas de diferenças finitas. O primeiro passo é a construção de uma malha representando todos os pontos Math input error da placa no instante de tempo Math input error. A malha adotada aqui para nossa solução é uniforme, ou seja, os pontos são igualmente espaçados considerando o tamanho total da malha no intervalo Math input error.

O segundo passo consiste no cálculo da temperatura na placa usando a equação de *Laplace*: Math input error, resultando na equação de calor por meio da aproximação por diferenças finitas a

seguir, sendo que Math input error é a representação da malha, Math input error e Math input error são os pontos dessa malha e Math input error é o passo.

Math input error

A Figura 6.15 apresenta essa solução de forma visual.

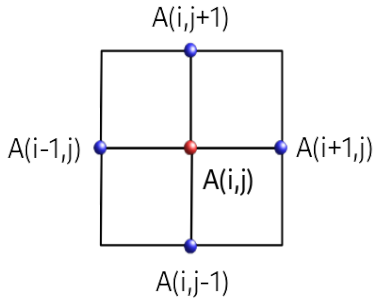

Figura 6.15: Malha do método Jacobi

Por fim, foi utilizado o método de Jacobi para resolver esse sistema de equação no qual os valores da malha na Math input error iteração são utilizados para calcular os novos valores na Math input error iteração. A solução sequencial para o método de Jacobi é apresentada no exemplo a seguir.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define COLUMNS    7000
#define ROWS       7000
#define MAX_ERROR 0.0001
double Anew[ROWS+2][COLUMNS+2];
double A[ROWS+2][COLUMNS+2];
void iniciar() {
    int i,j;
    // iniciacao dos valores da grade
    for (i = 0; i <= ROWS+1; i++) {
         for (j = 0; j <= COLUMNS+1; j++) {
              A[i][j] = 0.0;
         }
    }
    // iniciacao dos valores de bordas laterais
    for (i = 0; i <= ROWS+1; i++) {
         A[i][0] = 0.0;
         A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }
    // iniciacao dos valores de bordas superiores/inferiores
    for (j = 0; j <= COLUMNS+1; j++) {
         A[0][j] = 0.0;
         A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
int main(int argc, char *argv[]) { /* seq_jacobi.c  */
int i, j;
int max_iterations=1000;
int iteration=1;
double dt=100;
    iniciar();
```

```
    while (dt > MAX_ERROR && iteration <= max_iterations ) {
        for (i = 1; i <= ROWS; i++)
             for (j = 1; j <= COLUMNS; j++) {
                  Anew[i][j] = 0.25 * (A[i+1][j] +
                  A[i-1][j] + A[i][j+1] + A[i][j-1]);
             }
        dt = 0.0;
        for (i = 1; i <= ROWS; i++)
             for (j = 1; j <= COLUMNS; j++) {
                  dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
                  A[i][j] = Anew[i][j];
             }
        iteration++;
    }
    printf("\nErro maximo na iteracao %d era %f\n", iteration-1, dt);
    return(0);
}
```

O primeiro laço de repetição (while) no programa controla a convergência do método de Jacobi considerando uma quantidade máxima de iterações (*max_iterations*) e um erro máximo (*MAX_ERROR*). Internamente a este laço de repetição, existem dois outros laços (*for*) que realizam a computação do método.

No primeiro laço do *while* são calculados os novos valores para cada elemento da malha (matriz **Anew[]**) com base nos valores atuais de seus vizinhos (matriz **A[]**). A solução computacional da equação do calor está nesse laço e, portanto, deve-se percorrer todos os elementos da malha (por isso da existência de um outro laço aninhado).

No segundo laço do *while* é calculado o valor do erro máximo da solução a partir dos resultados da atualização da malha pelo método de Jacobi. Nesse laço também é atualizada a matriz (matriz **A[]**) com os valores atuais para a próxima iteração da solução.

Esse exemplo pode ser salvo com o nome *seq_jacobi.c* e sua compilação e execução é possível com os seguintes comandos, em um terminal de um sistema operacional tipo Linux:

```
$ gcc -o seq_jacobi seq_jacobi.c -O3 -std=c11 -lm
$ ./seq_jacobi
```

O resultado da execução desse programa é o seguinte:

```
Erro maximo na iteracao 1000 era 0.035831
```

Durante a paralelização deste código foi realizada uma série de otimizações no código-fonte original (algumas delas foram discutidas na Seção 6.10):

- As matrizes **A[]** e **Anew[]** foram transformadas em ponteiros para facilitar a atualização da malha;
- O segundo laço de repetição foi removido e o cálculo do erro máximo foi transferido para o primeiro laço de repetição;
- A atualização da malha atual é feita por meio de troca de ponteiros, evitando a cópia de todos os valores da malha;
- Para garantir que as matrizes sejam iguais, os valores de **Anew[]** são iniciados juntamente de **A[]**.

Uma possível forma de paralelizar esse código é ilustrada no próximo exemplo.

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#define MAX_ERROR 0.0001
size_t COLS=((size_t)7000);
size_t ROWS=((size_t)7000);
double **restrict A, **restrict Anew, **restrict tmp;
void iniciar();
int main(int argc, char *argv[]) { /* omp_jacobi.c  */
int i, j;
int max_iterations=1000;
int iteration=1;
double dt=100;
    A = (double **) malloc((ROWS+2) * sizeof(double*));
        for(int i = 0; i < (ROWS+2); i++)
            A[i] = (double *) malloc((COLS+2) * sizeof(double));
    Anew = (double **) malloc((ROWS+2) * sizeof(double*));
        for(int i = 0; i < (ROWS+2); i++)
            Anew[i] = (double *) malloc((COLS+2) * sizeof(double));
    iniciar();
    double t_inicio = omp_get_wtime();
    while (dt > MAX_ERROR && iteration <= max_iterations ) {
        dt = 0.0;
#pragma omp parallel for reduction(max:dt) default(none) private(j) shared(A, Anew, COLS,
ROWS)
        for (i = 1; i <= ROWS; i++)
            for (j = 1; j <= COLS; j++) {
                Anew[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1]);
                dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
            }
        tmp = Anew;
        Anew = A;
        A = tmp;
        iteration++;
    }
    double t_fim = omp_get_wtime();
    printf("Erro maximo na iteracao %d era %f\n", iteration-1, dt);
            printf("Tempo   paralelo:   \t   %f   com   %d   threads.\n",   t_fim-t_inicio,
omp_get_max_threads());
    free(Anew);
    free(A);
    return(0);
}
void iniciar(){
    int i,j;
    // iniciacao dos valores da grade
    for (i = 0; i <= ROWS+1; i++) {
        for (j = 0; j <= COLS+1; j++) {
            A[i][j] = 0.0;
            Anew[i][j] = 0.0;
        }
    }
    // iniciacao dos valores de bordas laterais
    for (i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        Anew[i][0] = 0.0;
        A[i][COLS+1] = (100.0/ROWS)*i;
```

```
        Anew[i][COLS+1] = (100.0/ROWS)*i;
    }
    // iniciacao dos valores de bordas superiores/inferiores
    for (j = 0; j <= COLS+1; j++) {
        A[0][j] = 0.0;
        Anew[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLS)*j;
        Anew[ROWS+1][j] = (100.0/COLS)*j;
    }
}
```

Devido ao método de Jacobi ser iterativo, o laço de controle da convergência não pode ser paralelizado, pois há uma dependência entre a Math input error iteração e a Math input error iteração. Contudo, o laço de repetição *for* para o cálculo dos valores da nova malha a partir dos valores atuais de seus vizinhos pode ser paralelizado.

Esse laço foi paralelizado por meio da diretiva **#pragma omp for**. A estratégia adotada prioriza a decomposição em blocos no sentido da linha, onde cada iteração paralelizada representa uma linha da malha. O escalonamento estático **schedule(static)** do OpenMP foi utilizado e cada *thread* recebe um intervalo de índices, sendo que esses índices de linhas são utilizados para o cálculo da nova malha. A quantidade de linhas que cada *thread* recebe é aproximadamente Math input error. O cálculo do erro máximo também foi alterado para uma cláusula de redução: **reduction(max:dt)**.

Por fim, a inversão dos ponteiros **Anew** e **A** pode ser feita visando o início de uma nova iteração do laço *while*. Com a otimização, não é necessário copiar os valores entre essas matrizes, reduzindo ainda mais o tempo de computação.

Essa solução pode ser salva com o nome *omp_jacobi.c*. A compilação, a execução utilizando todas as *threads* disponíveis, e o resultado da execução desse programa é visto a seguir. Esses comandos foram realizados em um terminal de um sistema operacional tipo Linux:

```
$ gcc -o omp_jacobi omp_jacobi.c -fopenmp -O3 -std=c11 -lm
$ ./omp_jacobi
Erro maximo na iteracao 1000 era 0.035831.
Tempo paralelo:          14.301633 com 48 threads.
```

Avaliação de desempenho

A análise de escalabilidade desse problema é bem mais desafiadora, mesmo que os experimentos sejam realizados em um bom equipamento de teste. Neste caso, escolhemos uma arquitetura de multiprocessamento simétrico com um total de 24 processadores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz).

Tabela 6.7: Métricas da solução otimizada - Jacobi

***Threads* p**	**Tempo de execução**	***Speedup***	**Eficiência**
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

***Threads* p**	**Tempo de execução**	***Speedup***	**Eficiência**
Math input error	Math input error s	Math input error	Math input error
Math input error	Math input error s	Math input error	Math input error

Para efeito de comparação entre o tempo de execução da solução sequencial e das otimizações realizadas na solução paralela, essas soluções foram executadas, obtendo-se o tempo de Math input error para a solução sequencial e o tempo de Math input error para a solução otimizada com uma *thread*. Os resultados dos experimentos com a solução otimizada foram coletados e são apresentados na Tabela 6.7. Esses tempos foram utilizados nos cálculos das métricas conforme discutido na Seção 2.4.1.

Para entender o comportamento da solução, desenhamos as curvas de escalabilidade na Figura 6.16. Nesse gráfico, a reta contínua representa o *speedup* linear ideal; a reta anotada com pontos indica o *speedup* da solução otimizada.

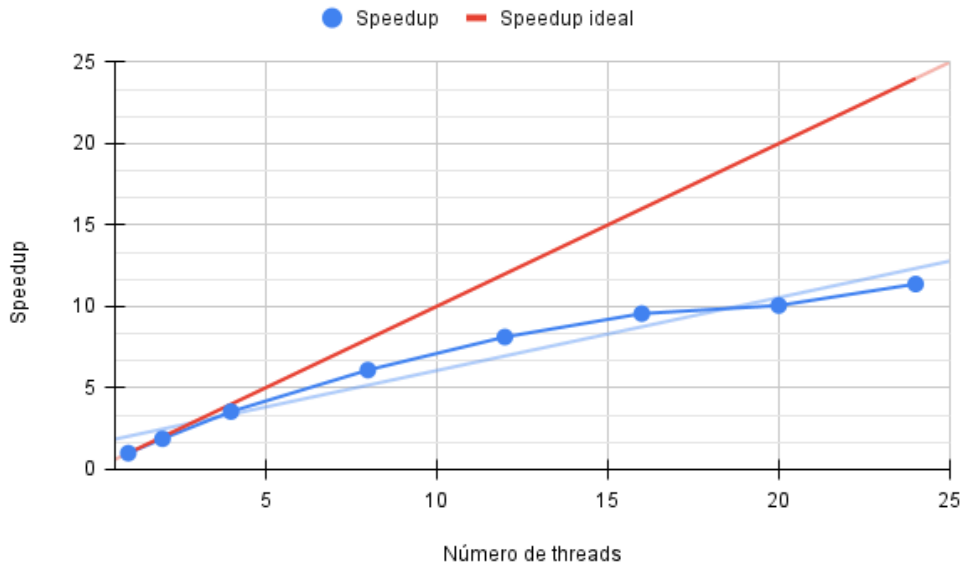

Figura 6.16: Gráfico de *speedup* - método de Jacobi

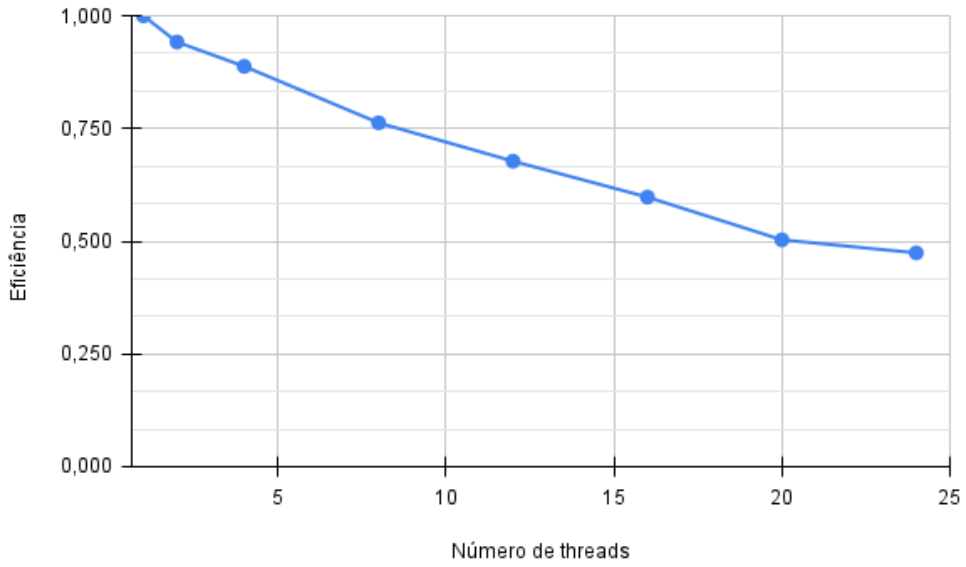

Figura 6.17: Gráfico de eficiência - método de Jacobi

Apesar de o estudo inicial indicar que a solução do problema apresentaria bons resultados, os dados coletados do experimento mostraram o contrário. A partir da utilização de duas *threads*, o desempenho cai consideravelmente e estabiliza o *speedup* por volta de 10,0. Essa queda, como acontece com muitos programas paralelos em memória compartilhada, se dá provavelmente em função de uma baixa localidade dos dados carregados nas linhas do cache.

Analisando o comportamento da solução por meio da Figura 6.15, os acessos que acontecem nas posições Math input error e Math input error, além da computação realizada na linha em Math input error e Math input error, aumentam a taxa de falhas da memória cache, causando essa diminuição de desempenho. Esse impacto é percebido também no gráfico de eficiência, como mostra a Figura 6.17.

Realizamos também a medição da escalabilidade de acordo com a definição de Gustafson (veja a Seção 2.4.3), quando aumentamos proporcionalmente a quantidade de células da matriz com o número de processadores. Além disso, mantivemos o erro máximo (*MAX_ERROR*=Math input error) e a quantidade de iterações (1.000). Dessa forma, a matriz original de dimensão Math input error com Math input error células passaria a ter Math input error células e dimensão aproximada de 9.900 para o experimento com 2 *threads*. As demais dimensões das matrizes utilizadas neste experimento, juntamente às medições de tempo de execução e às métricas de *speedup* e eficiência podem ser vistas na Tabela 6.8.

Tabela 6.8: Medição da solução do método de Jacobi - Gustafson

Tamanho	**Tempo Seq.**	**Threads (*p*)**	**Tempo Paralelo**	**Speedup**	**Efici- ência**
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

Tamanho	Tempo Seq.	Threads (p)	Tempo Paralelo	Speedup	Efici- ência
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error
Math input error	Math input errors	Math input error	Math input errors	Math input error	Math input error

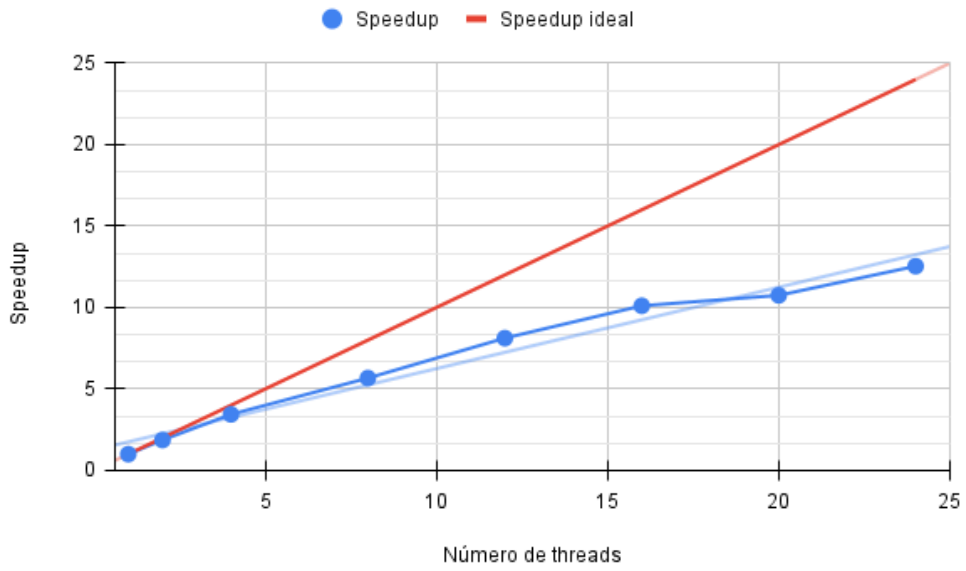

Figura 6.18: Gráfico de *speedup* - método de Jacobi

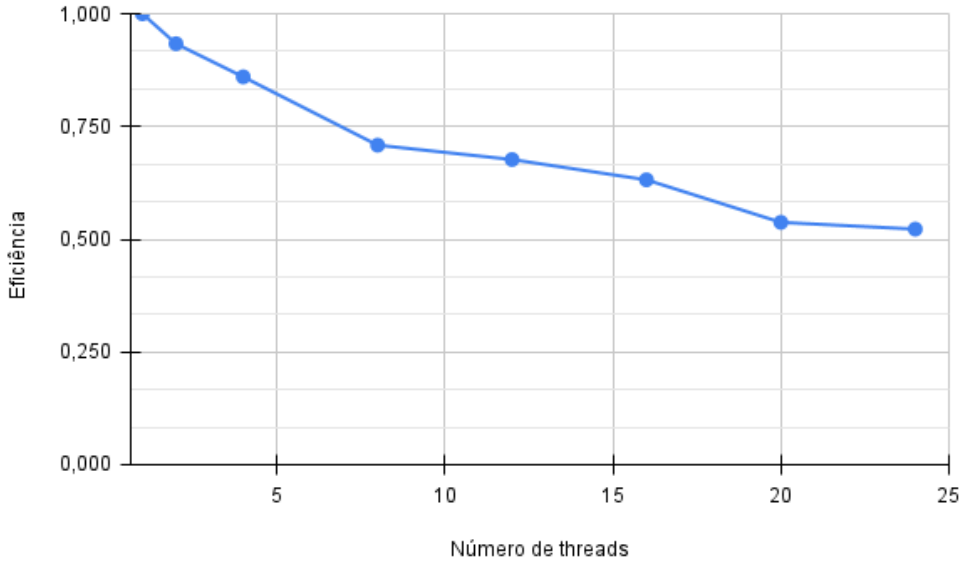

Figura 6.19: Gráfico de eficiência - método de Jacobi

Desenhamos o gráfico da escalabilidade da solução melhorada na Figura 6.18. Essa linha se manteve semelhante à anterior (da Figura 6.16) e a partir da utilização de duas *threads*, o *speedup* também cai e se estabiliza por volta de 10,5. O mesmo comportamento de queda pode ser observado no gráfico de eficiência da Figura 6.19.

Esses resultados apontam que os dois métodos de avaliação, Amdahl e Gustafson, são aproximadamente equivalentes para esse tipo de problema, tendo as mesmas deficiências de desempenho. Ressalve-se também o fato de que nem sempre é possível obter-se bons resultados com a paralelização, dados os baixos valores de *speedup* e eficiência apresentados neste exemplo.

6.9 Exercícios

1. Com relação ao OpenMP respondas às seguintes questões:
    1. Para quais tipos de arquitetura se destina o padrão OpenMP?
    2. Explique o funcionamento das regiões paralelas no OpenMP?
    3. Qual a utilidade da macro _OPENMP?
2. Qual a diferença e qual a função das seguintes diretivas do OpenMP:
    1. **parallel**;
    2. **single**;
    3. **sections**;
    4. **for**.
3. Quais diretivas podem ser combinadas com a diretiva **parallel**?
4. Qual a precedência e quais os métodos para determinar o número de *threads* de uma região paralela?

5. Qual a diferença e qual o valor retornado pelas seguintes funções do OpenMP:
    1. **omp_get_num_threads()**;
    2. **omp_get_max_threads()**;
    3. **omp_get_thread_num()**;
    4. **omp_get_num_procs()**.
6. Quais as vantagens do uso da rotina **omp_get_wtime()** em relação às rotinas padrão de tempo da linguagem C?
7. Com quais diretivas a cláusula **num_threads** pode ser utilizada?
8. Quais as condições que uma variável não listada na cláusula **shared** deve obedecer quando a cláusula **default(none)** é utilizada?
9. Qual cláusula é utilizada para dar valor inicial a uma variável local em uma região paralela?
10. Modifique o exemplo da Seção 6.4.1 para a cláusula **firstprivate** utilizando no seu lugar a cláusula **private**. O que muda na execução do programa?
11. Enumere e descreva quatro tipos de operação de redução utilizados na cláusula **reduce** do OpenMP?
12. Explique o funcionamento da cláusula **copyprivate** na diretiva **single**.
13. Qual a finalidade da cláusula **lastprivate**? Para quais diretivas ela pode ser utilizada?
14. Paralelize o exemplo a seguir sem o uso da diretiva **parallel for** ou **for**.

```
for (i = 0; i < n; i++)
     z[i] = a * x[i] + y;
```

15. Explique as diferenças entre os escalonamentos STATIC, DYNAMIC, GUIDED, AUTO e RUNTIME.
16. No código a seguir, faça um diagrama do escalonamento das iterações do laço entre 4 *threads*.

```
#pragma omp for schedule(guided, 5)
        for (j = 0; j < 50; j++)
             a[j]= b[j] + c[j];
```

17. Qual a finalidade e quando pode ser utilizada a cláusula **ordered** em um laço?
18. Qual a finalidade e quando pode ser utilizada a cláusula **collapse** em um laço?
19. Explique por que cada um dos laços a seguir pode ou não pode ser paralelizado com a diretiva **parallel for**.

    a)

```
for (i = 0; i < N ; i ++)
         if (x [i] > maxval) break;
```

b)

```
 for (i = 0; i < N ; i ++)
          for (j = 0; j < i ; j++)
               a[j][i] = a[j+1,i];
```

c)

```
 for (i = 0; i < N ; i ++) {
          if (weight [i].HEAVY) {
              pid = fork();
              if (pid == -1 ) {
                  perror("fork");
                  exit (1);
              }
              if (pid = = 0 ) {
                  heavy_task( );
                  exit (1);
              }
          else
              light_task ();
}
```

d)

```
    for (i = 0; i < N ; i ++) {
         a[i] = a[i] * a[i];
         if (fabs(a[i]) > machine_max||fabs(a[i]) < machine_min) {
             printf ("i = %d \n", i);
             break;
         }
    }
```

20. Para que serve uma diretiva **barrier**? Quais diretivas possuem uma barreira implícita no seu final?

21. Quais as cláusulas da diretiva **master**? Quais as diferenças para a diretiva **single**?

22. Reescreva o código seguinte, colocando as cláusulas **nowait** explicitamente, de modo a maximizar o desempenho, mas sem alterar o resultado da execução.

```
#pragma omp parallel
{
    #pragma omp for
    for (j = 0; j < n; j++)
        a[j]= b[j] + c[j];
    #pragma omp for
    for (j = 0; j < n; j++)
        d[j] = e[j] * f;
    #pragma omp for
    for (j = 0; j < n; j++)
        z[j] = (a[j]+a[j+1]) * 0.5;
}
```

23. Qual a finalidade da diretiva **critical**? Para que servem os nomes na diretiva **critical**?

24. Qual a finalidade da diretiva **atomic**? Quais as cláusulas que admite e qual a sua finalidade?

25. O que é mais eficiente? Utilizar a diretiva **critical** ou **atomic**? Explique.

26. Qual a finalidade das funções de **lock()**? Quando devem ser utilizadas?

27. O que é uma variável de chave *lock*? Qual o tipo que deve ser utilizado para declará-la?

28. Reescreva o exemplo da Seção 6.5.4 com uso de funções de **lock()**.

29. Generalize o exemplo a seguir para um número arbitrário de *threads*, sem uso da diretiva **for** ou **parallel for**. Considere que o número de colunas *m* é maior que o número de *threads* em execução.

```
#pragma omp parallel default(none) private(i,tid) shared(a,n)
    tid = omp_get_thread_num();
    for (i = 0; i < n; i++)
        a[i][tid] = 1.0;
```

30. Considere o seguinte laço:

```
    x= 1;
    #pragma omp parallel for firstprivate (x)
    for (i = 0; i < n; i++) {
        y[i] = x + i;
        x = i;
    }
```

a. Por que esse laço está incorreto? y [i] recebe o mesmo resultado independente do número de *threads* executando o laço?

b. Qual o valor da variável i ao final do laço? Qual é o valor da variável x ao final do laço?

c. Qual seria o valor de x ao final do laço se os eu escopo fosse shared?

d. Esse laço pode ser paralelizado corretamente (isto é, preservando a semântica sequencial) apenas com o uso de diretivas OpenMP?

31. Paralelize o seguinte trecho de programa com uso de diretivas OpenMP e apresente uma solução para a condição de corrida existente.

```
void Histogram(PPMImage *image, float *h) {
int i, j,  k, l, x, count;
int rows, cols;
float n = image->y * image->x;  /* Total de pixels da imagem */

    cols = image->x;
    rows = image->y;
        /* Faz a leitura de cada pixel, normalizando entre 0 e 3 */
    for (i = 0; i < n; i++) {
        image->data[i].red = floor((image->data[i].red * 4) / 256);
        image->data[i].blue = floor((image->data[i].blue * 4) / 256);
        image->data[i].green = floor((image->data[i].green * 4) / 256);
    }
       /* Faz a montagem do histograma */
       count = 0;
       x = 0;
       /* Para cada um dos 64 pontos do histograma ... */
       for (j = 0; j <= 3; j++) {
        for (k = 0; k <= 3; k++) {
            for (l = 0; l <= 3; l++) {
    /* ... percorre todos os pixels da imagem */
```

```
                for (i = 0; i < n; i++) {
                    if (image->data[i].red == j &&
                    image->data[i].green == k &&
                    image->data[i].blue == l) {
                        count++;
                    }
                }
    /* Monta o histograma percentual com 64 pontos */
                h[x] = count / n;
                count = 0;
                x++;
            }
        }
}
```

---

1. Veja mais detalhes em https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html↩

7 OpenACC

Neste capítulo estudaremos o OpenACC, um modelo de programação para computação paralela aberto desenvolvido com o objetivo de simplificar a programação paralela, oferecendo alto desempenho e portabilidade entre diversos tipos de arquiteturas: *multicore*, *manycore* e GPUs (OpenACC-Standard.org 2019).

7.1 Modelo de programação OpenACC

As arquiteturas *multicore*, *manycore* e GPUs têm alcançado um alto grau de paralelismo com a utilização de um número cada vez maior de processadores. Essas arquiteturas, com uso de processamento paralelo maciço, têm aplicação nas áreas de geofísica, sequenciamento genético, simulações de modelos matemáticos e previsão do tempo, entre outras (Silva 2018), (Costa, Silva, and Teixeira 2018).

O OpenACC é compatível com os modelos de programação OpenMP e MPI, sendo que ambas abordagens podem ser combinadas com o OpenACC. Em geral, as diretivas do OpenACC são muito semelhantes às do OpenMP. Em relação ao CUDA, o OpenACC é totalmente compatível, o que torna a necessidade de alteração do código a menor possível.

O modelo de programação OpenACC possui algumas características que o tornam fácil e simples de utilizar, como:

- É independente de fabricante;
- Oculta a complexidade do *hardware* dos programadores;

- Requer poucas modificações ao código-fonte;
- É mais fácil de programar e depurar que o CUDA;
- Possui algumas facilidades que o CUDA não oferece;
- O mesmo código por ser usado em *multicore, manycore* e GPUs;
- É similar ao OpenMP (familiaridade);
- Fácil transição para o OpenMP 4.5.

Em um processamento convencional, as *threads* são executadas por processadores escalares. No OpenACC, a maior parte da aplicação é executada no hospedeiro, em um processador convencional, mas a execução dos laços computacionalmente intensivos (chamados de *kernels*) é feita em um dispositivo acelerador, normalmente uma GPU, e controlada a partir do hospedeiro (CPU). Isto é, a transferência e execução dos *kernels* no acelerador, a transferência de dados de/para o acelerador, a alocação de memória no acelerador, a transferência de resultados de volta para o hospedeiro e a desalocação de memória são todas gerenciadas pelo hospedeiro. Na maioria dos casos, o hospedeiro pode enfileirar uma sequência de operações a serem executadas no dispositivo acelerador, uma após a outra.

A GPU é construída em torno de um arranjo escalável de multiprocessadores de fluxo (Streaming Multiprocessor - SM) capazes de executar um conjunto de *threads* simultaneamente, que recebem o nome de "blocos de *thread*". Um laço de programa é transformado em uma grade de *kernel*, contendo vários blocos de *threads*, que são numerados e distribuídos para os SM com capacidade de execução disponível. Um bloco de *threads* pode ser executado concorrentemente em um multiprocessador (SM). Assim que um bloco de *thread* termina a sua execução, novos blocos são

lançados para os multiprocessadores (SMs) ociosos. Veja um esquema comparativo na Figura 7.1[1].

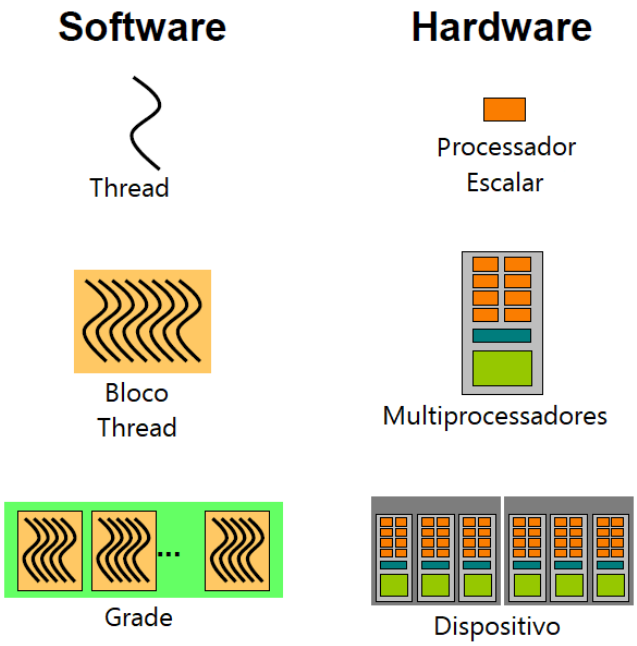

Figura 7.1: Modelo de Execução

Ao se transportar um código sequencial para um modelo com uso do OpenACC, deve-se observar que normalmente os aceleradores e as CPUs não compartilham a mesma memória. Em outras palavras, o acelerador não tem acesso direto à memória da CPU. A memória da CPU costuma ser maior, porém é mais lenta que a memória do acelerador. Para usar a GPU os dados devem ser transferidos da memória principal, no hospedeiro, para a memória da GPU através de barramentos de E/S, cuja largura de banda (velocidade) é muito menor do que a velocidade de acesso das GPUS e da CPUs às suas respectivas memórias. Isso significa que o gerenciamento das transferências de dados entre as memória da GPU e da CPU é de capital importância para um bom desempenho da aplicação escrita em OpenACC.

Antes de iniciar o processo de programação utilizando o OpenACC, é recomendável realizar uma análise do código a ser paralelizado,

seguindo o ciclo de desenvolvimento e análise de código conforme descrito a seguir:

Identificar o paralelismo: Realizar o *profile* do código para entender onde o programa está gastando mais tempo e quanto paralelismo está disponível para ser acelerado nas rotinas mais importantes. Os aceleradores têm melhor desempenho quando há um quantidade significativa de paralelismo para ser explorado, de modo que se deve procurar por laços e laços aninhados com um alto número de iterações independentes.

Expressar o paralelismo: Colocar diretivas OpenACC nos laços identificados no primeiro passo indicando ao compilador onde paralelizar. Programar com OpenACC consiste em dar ao compilador informação suficiente para acelerar com efetividade o código, então durante este passo adicione diretivas em todos os laços onde se acredite ser possível mover a computação para o acelerador.

Expressar localidade de dados: O compilador precisa saber não apenas qual código paralelizar, mas também quais dados serão necessários no acelerador para aquele código. Depois de expressar o paralelismo disponível, é possível que o código obtido fique mais lento. Como explicaremos mais tarde, essa lentidão ocorre porque o compilador toma decisões cautelosas sobre os dados a serem movidos para o acelerador. Durante este passo, deve-se expressar ao compilador o seu conhecimento de quando e onde os dados são realmente necessários no acelerador.

Otimização: O compilador usualmente faz um bom trabalho em acelerar o código, mas algumas vezes é possível obter um desempenho melhor dando ao compilador um pouco mais de informação a respeito dos laços; pela reestruturação do código para aumentar o paralelismo; ou melhorando o padrão de acesso aos dados. Na maior parte do tempo isso leva a pequenas melhorias, mas às vezes os ganhos podem ser maiores.

Na Figura 7.2 é apresentado o ciclo de desenvolvimento e análise do código.

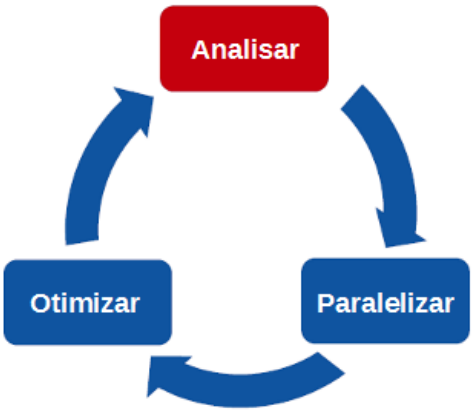

Figura 7.2: Ciclo de desenvolvimento e análise do código

É realmente importante depois de cada um desses passos verificar se o programa ainda produz resultados corretos, pois é muito mais fácil identificar um erro logo depois de uma pequena mudança no código, do que após várias mudanças. Esse processo é um ciclo, então depois de otimizar um trecho de código, deve-se voltar para o início e encontrar o próximo trecho para expressar paralelismo.

Algumas observações em relação ao código a ser paralelizado são: a computação deve ser baseada em laços aninhados processando arranjos (matrizes e vetores), sendo que esses laços devem ter um número de iterações definido, seja em tempo de compilação ou de execução. Os laços baseados em estruturas do tipo *while* são difíceis de serem paralelizados no OpenACC, porque a sua execução não é feita com facilidade nas GPUS.

As estruturas de dados utilizadas devem ser arranjos simples, sendo que tipos de dados derivados, arranjos de ponteiros, listas encadeadas podem estar além da capacidade de paralelização do compilador. Os laços aninhados devem ter um grande número de iterações, pelo menos na ordem de alguns milhares, quanto mais, melhor. Embora laços com menor número de iterações também

sejam executados no acelerador, a eficiência será sempre muito pequena.

Os laços devem abranger a maior quantidade possível de código, a partir do nível mais alto possível. O uso de um ou dois comandos *if* dentro dos laços aninhados é aceitável, contudo, um número maior do que isso levará a uma redução acentuada de desempenho na maioria dos aceleradores.

7.1.1 Níveis de paralelismo do OpenACC

O OpenACC é uma linguagem para aceleradores genéricos com três níveis de paralelismo, que especificam o nível de paralelismo contido em uma região paralela, e são chamados de gangue (*gang*), trabalhador (*worker*), e vetor (*vector*). Esse tipo de paralelismo é mais útil quando estamos trabalhando com laços multidimensionais aninhados, ou seja, com duas ou mais variáveis de iteração. Adicionalmente, é possível também marcar a execução de um laço como sequencial (*seq*).

Quando um laço é paralelizado, o nível mais alto e de maior granularidade de paralelismo é uma gangue. Quando uma diretiva **parallel** ou **kernels** é encontrada, várias gangues serão criadas e as iterações dos laços dentro da região são distribuídas através das gangues. As gangues trabalham de forma independente umas das outras e não há nenhum modo de determinar exatamente quantas gangues estão executando em um dado momento. Em muitas arquiteturas as gangues podem ter memórias completamente separadas.

Uma gangue é composta por um ou vários trabalhadores, com paralelismo de granularidade mais fina, sendo que todos os trabalhadores de uma gangue podem compartilhar os mesmos recursos, como memória cache ou o mesmo multiprocessador de fluxo (SM).

Os trabalhadores então computam um vetor, sendo que os vetores trabalham de forma sincronizada (*lockstep*) usando paralelismo SIMD/SIMT. Um vetor é o nível mais baixo de paralelismo, e cada gangue vai ter pelo menos um vetor. Cada vetor tem a habilidade de executar uma única instrução em vários elementos de dados. O OpenACC procurar defini-los de uma maneira genérica, sem vinculá-los a um tipo específico de *hardware*.

Assim, um único laço pode estar marcado para execução por um, dois ou todos os três níveis de paralelismo: gangue, trabalhador ou vetor; e as iterações do laço são distribuídas entre as gangues, trabalhadores e vetores da forma apropriada.

É possível também combinar essas cláusulas em um laço específico. Por exemplo, um laço pode ser particionado entre gangues, cada uma delas com um trabalhador definido implicitamente, e depois vetorizado, como exemplificado na Figura 7.3.

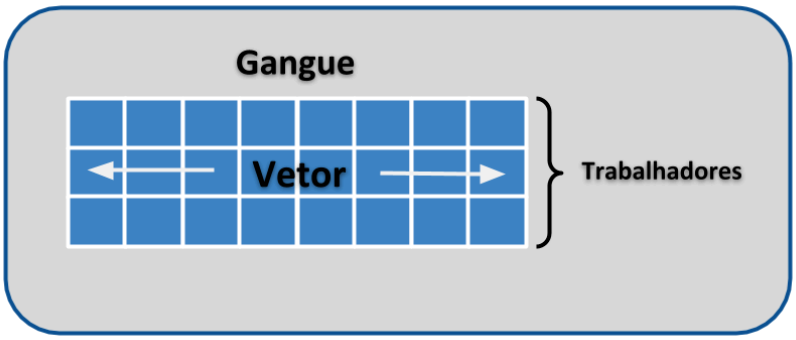

Figura 7.3: Gangues x Trabalhadores x Vetor

A especificação OpenACC reforça que o laço mais externo deve ser um laço de uma gangue, o laço paralelo mais interno deve ser um laço vetor e um laço trabalhador pode aparecer no meio, mas usualmente é omitido. Um laço sequencial (cláusula **seq**) pode aparecer em qualquer nível. Veremos mais detalhes na Seção 7.5.

De outra forma, podemos dizer que a execução de um *kernel* utiliza um conjunto de *threads* que são mapeadas para os recursos de

hardware do acelerador. Essas *threads* são agrupadas em times com o mesmo tamanho, com uma *thread master* por time, o que define uma gangue. Cada time é organizado em uma malha bidimensional de trabalhadores e vetores. Um trabalhador é na realidade um vetor de comprimento *vector_length threads*. Então o número total de *threads* é dado pela equação:

num_threads = num_gangs * num_workers * vector_length

Os níveis de paralelismo usados no OpenACC são comparáveis aos níveis de execução usados na programação CUDA. São ideias parecidas, mas a sua terminologia é um pouco diferente. No CUDA, uma execução em paralelo é organizada em grades (*grids*), blocos de *threads* (*thread blocks*) e *threads*, executadas em um elemento processador (PE). Uma gangue é composta por um ou vários trabalhadores e corresponde a um bloco de *threads* em CUDA. Em uma GPU, o elemento processador(PE) é um multiprocessador de fluxo (SM). Um conjunto de gangues OpenACC (blocos de *thread* CUDA) mapeiam para diversos PEs.

Cada trabalhador corresponde a um número de threads igual ao tamanho do vetor. Então um trabalhador corresponde a uma *warp* em CUDA apenas se o vetor tiver um comprimento igual a 32; já que um trabalhador não corresponde necessariamente a uma *warp*. Por exemplo, um trabalhador corresponderá a duas *warps* se o vetor tiver tamanho 64. A característica principal de uma *warp* é a execução de todas as suas *threads* concorrentemente.

Uma *grade* CUDA é composta de vários blocos de *threads*, ou gangues do OpenACC, os quais podem ser organizados em uma ou duas dimensões. Não existe um elemento específico correspondente à *grade* CUDA no OpenACC.

Os aceleradores podem ter limitações em relação aos valores a serem atribuídos a esses particionamentos. Por exemplo, para GPUS da NVIDIA, existem as seguintes limitações:

- O comprimento de um vetor deve ser um múltiplo de 32 (até um máximo de 1024);

- O tamanho de uma gangue é dado pelo número de trabalhadores vezes o tamanho de um vetor, não podendo também ser maior que 1024.

Já no caso da GPU Radeon da AMD, que tem uma estrutura bem diferente das GPUs NVIDIA, os seguintes valores se aplicam:

- O comprimento de um vetor deve ser um múltiplo de 64 (até um máximo de 2048);

- O tamanho de uma gangue é dado pelo número de trabalhadores vezes o tamanho de um vetor, não podendo ser maior também que 2048.

Contudo, os valores máximos apresentados para a GPU da AMD podem não ser suportados pela implementação atual do compilador em uso, podendo haver ainda outros limites para o número máximo de gangues e de trabalhadores.

7.2 Diretivas principais

O modelo de programação usado no OpenACC é baseado em diretivas e cláusulas. As diretivas são comandos passados pelo programador ao compilador, que eventualmente são acompanhadas de parâmetros adicionais chamados de cláusulas.

As diretivas do OpenACC são muito parecidas com as diretivas do OpenMP e também são escritas na forma de **pragma**. Uma das vantagens no uso de diretivas, por exemplo, acontece no caso de o código precisar de pequenas modificações, que podem ser feitas de forma incremental, um **pragma** de cada vez. Esse processo torna-se especialmente útil para fins de depuração, já que, ao se fazer uma única alteração, é possível identificar rapidamente um erro.

É possível desativar o uso do OpenACC em tempo de compilação. Quando isso acontece, o **pragma** referente ao OpenACC é considerado um comentário, sendo ignorado pelo compilador. Com isso, pode-se usar um único código-fonte para gerar tanto um código com suporte a acelerador quanto uma versão sequencial.

O OpenACC pode ser utilizado em códigos escritos em linguagens de programação C/C++ ou Fortran, devendo-se atentar para a sintaxe correta de acordo com o tipo de linguagem de programação correspondente, conforme visto a seguir. Neste livro, vamos apresentar sintaxe e exemplos apenas na linguagem C.

```
#pragma acc data [cláusula]
{
      região de código ...
}

!$acc data
    região de código ...
!$acc end data
```

7.2.1 Diretiva kernels

A primeira diretiva apresentada será a diretiva **kernels**, utilizada para informar ao compilador as regiões do código potencialmente paralelizáveis. O compilador será o responsável por identificar exatamente quais são as regiões e qual será a estratégia utilizada. A estratégia definida pelo compilador pode ser extrair o máximo de paralelismo do código ou executar somente o mínimo de paralelismo. Sua sintaxe é a seguinte:

#pragma acc kernels [cláusulas]

Com uso da diretiva **kernels**, o código será analisado pelo compilador e só será paralelizado quando houver certeza de que é seguro fazê-lo. Em alguns casos, pode não haver informações suficientes para o compilador determinar se é seguro paralelizar um

laço, então essa paralelização não será feita. Os passos do processo de compilação do código usando a diretiva **kernels** são:

- Analisar o código para identificar regiões de paralelismo;
- Se encontrado, identificar quais dados devem ser transferidos;
- Criar um *kernel*;
- Movimentar os dados para o dispositivo acelerador.

```
#include <stdlib.h>
void saxpy(int n, float a, float * restrict x, float * restrict
y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
         y[i] = a*x[i] + y[i];
}
int main(int argc, char *argv[]) { /* acc_saxpy.c   */
int n = 1<<20; // 1 milhão de floats
float *x = (float*)malloc(n*sizeof(float));
float *y = (float*)malloc(n*sizeof(float));
#pragma acc kernels
    for (int i = 0; i < n; ++i) {
         x[i] = 2.0f;
         y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    free(x);
    free(y);
    return(0);
}
```

No exemplo apresentado, o uso da diretiva **kernels** apenas sugere que esse é um laço paralelo, cabendo ao compilador tentar gerar um *kernel* paralelo para ser executado no acelerador. Note que não houve necessidade de alocar ou iniciar vetores no acelerador; não houve necessidade de indicar quais dados deveriam ser movidos de/para o acelerador; não houve necessidade de se reescrever o

código do laço original; nem houve necessidade de se lançar explicitamente o *kernel* para execução no acelerador. O compilador OpenACC e o ambiente de execução cuidam de tudo isso de forma transparente para o programador.

Em alguns casos, como no código C/C++, onde arranjos são passados para funções como ponteiros, o compilador nem sempre consegue determinar se dois arranjos compartilham ou não a mesma área de memória, o que é conhecido como *aliasing* de ponteiros. Caso o compilador não possa identificar que dois ponteiros não possuem *aliasing*, ele não será capaz de paralelizar um laço que acessa esses arranjos.

Uma prática recomendada para programadores em C é usar a palavra-chave *restrict* (ou o decorador *__restrict* em C++) sempre que possível, para informar ao compilador que os ponteiros não têm *aliasing*. Isso normalmente fornecerá ao compilador informações suficientes para paralelizar laços que de outro modo seriam executados sequencialmente.

Se o programador violar essa declaração, o comportamento do programa é indefinido. Quem programa deve garantir que os valores escritos pelos ponteiros não afetarão os valores lidos por qualquer outro ponteiro existente no mesmo contexto.

Cuidado para utilizar corretamente a palavra-chave, não confundir *float restrict *ptr* (incorreto) com *float *restrict ptr* (correto).

Além da palavra-chave *restrict*, variáveis podem ser declaradas como constantes usando a palavra-chave *const*, o que permite ao compilador armazenar essa variável em memória apenas de leitura, caso esse tipo de memória exista no acelerador. O uso de *const* e *restrict* é uma boa prática de programação em geral, pois fornece ao compilador informações adicionais para serem utilizadas na otimização do código.

Um benefício adicional fornecido pela construção **kernels** é que, se os dados forem movidos para o dispositivo para uso em laços contidos na região, esses dados permanecerão no dispositivo por toda a extensão da região ou até que sejam necessários novamente no hospedeiro.

7.2.2 Diretiva parallel

A paralelização usando a diretiva **parallel** identifica uma região de código para ser paralelizada. O uso da cláusula **parallel** é um caminho mais fácil para a adaptação, por exemplo, de código escrito para o OpenMP.

#pragma acc parallel [cláusulas]

Quando o ambiente de execução encontra uma diretiva **parallel**, uma ou mais gangues são criadas para executar a região paralela no acelerador. O número de gangues, o número de trabalhadores por gangue e o comprimento do vetor de cada trabalhador permanece o mesmo até o final daquela região paralela.

A diretiva **parallel** informa ao compilador para criar uma região paralela, mas diferentemente da região **kernels**, o código na região paralela será executado de forma redundante por todas as gangues. Ou seja, não haverá nenhum compartilhamento de tarefas!

Dentro da região paralela os laços que serão paralelizados precisam ser indicados no código pelo programador. O uso da diretiva **loop** faz o compilador gerar uma versão paralela do laço para o acelerador. Se a diretiva **loop** não for especificada, a paralelização dos laços não será feita.

```
#include <stdlib.h>
void saxpy(int n, float a, float * restrict x, float *restrict
y)
{
    #pragma acc parallel
```

```
    #pragma acc loop
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
int main(int argc, char *argv[]) { /* acc_saxpy2.c   */
    int n = 1<<20; // 1 milhão de floats
    if (argc > 1)
        n = atoi(argv[1]);
    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    #pragma acc parallel
    #pragma acc loop
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    return 0;
}
```

O exemplo da Seção 7.2.1 é reapresentado agora com uso da diretiva **parallel**. Note que, caso não seja utilizada uma cláusula **async**, haverá uma barreira implícita no final da região paralela, ou seja, a execução da *thread* local só vai prosseguir quando todas as gangues chegarem ao final da região paralela.

Existem algumas restrições que destacamos em relação ao uso da diretiva **parallel**: a execução do programa não pode desviar para dentro ou para fora de uma região paralela; e o resultado do programa não pode depender da ordem de avaliação de suas cláusulas, ou de qualquer efeito colateral dessas avaliações.

7.2.3 Diretiva kernels vs diretiva parallel

Um dos maiores pontos de confusão para os novos programadores de OpenACC são as diretivas **parallel** e **kernels**, que parecem fazer a mesma coisa. Embora estejam fortemente relacionadas, existem diferenças sutis entre elas, com características distintas para cada

tipo de aplicação, sendo usadas de acordo com a necessidade de execução do código.

Existem códigos de fácil modificação, obtendo-se um melhor desempenho com o uso da diretiva **parallel**. Existem, porém, códigos cuja alteração é difícil, e o uso da diretiva **parallel** não é possível. Nesses casos é utilizada a diretiva **kernels**, pois as mudanças no código são as mínimas possíveis.

É possível cobrir um grande trecho de código com apenas uma única diretiva **kernels**. Essa construção fornece ao compilador uma margem de manobra maior para paralelizar e otimizar o código da forma mais adequada para o tipo de acelerador utilizado, mas também depende muito da capacidade do compilador de paralelizar automaticamente o código. Como resultado, é possível haver diferenças de desempenho e de nível de otimização em compiladores diferentes.

O uso diretiva **parallel** é uma afirmação do programador indicando que é seguro e desejável paralelizar a região afetada. Isso depende da identificação correta do paralelismo no código e remoção de construções no código que não sejam seguras para paralelizar. Se o programador afirmar incorretamente que um laço pode ser paralelizado, há uma grande chance de que o programa resultante produza resultados incorretos.

Em outras palavras: a construção **kernels** é como uma dica sobre onde o compilador deve procurar paralelismo, enquanto a diretiva **parallel** é uma afirmação para o compilador onde há paralelismo. Vejamos os exemplos:

```
#pragma acc kernels
{
    for (i = 0; i < n; i++)
         a[i] = 3.0f*(float)(i+1);
    for (i = 0; i < n; i++)
         b[i] = 2.0f*a[i];
}
```

```
#pragma acc parallel
{
#pragma acc loop
    for (i = 0; i < n; i++)
         a[i] = 3.0f*(float)(i+1);
#pragma acc loop
    for (i = 0; i < n; i++)
         b[i] = 2.0f*a[i];
}
```

Os exemplos listados possuem dois laços comparando o processo de paralelização utilizando as diretivas de **kernels** e **parallel**. Note que no processo de execução usando a diretiva **kernels** são gerados dois *kernels*, um para cada laço. Existe uma barreira entre os laços, desse modo o segundo laço só será iniciado quando o primeiro terminar.

No processo de execução usando a diretiva **parallel** é gerado um único *kernel*, eventualmente resultando na execução simultânea dos laços, o que não é recomendável, como veremos a seguir.

7.2.4 Diretivas parallel loop e kernels loop

As diretivas combinadas do OpenACC **parallel loop** e **kernels loop** são atalhos para especificar uma diretiva **loop** imediatamente após uma diretiva **parallel** ou **kernel**. O resultado equivale a se especificar explicitamente uma diretiva **parallel** ou **kernels** contendo uma diretiva **loop**. Qualquer cláusula permitida em uma diretiva **parallel** ou na diretiva **loop** é permitida na diretiva **parallel loop**. Qualquer cláusula permitida em uma diretiva **kernels** ou na diretiva **loop** é permitida na diretiva **kernels loop**. Um breve exemplo desse uso é listado a seguir, sendo que um detalhamento maior dessas cláusulas será apresentado nas seções seguintes.

```
#pragma acc kernels loop
    for (int i = 0 ; i < N; i++)
        a[i] = 0;
#pragma acc kernels loop
```

```
    for (int j = 0 ; j < M; j++)
         b[j] = 0;
```

Então, para se fazer o paralelismo de vários laços, deve-se acompanhar cada laço de uma diretiva **parallel loop** ou **kernels loop**. Isso é uma afirmação do programador para o compilador indicando que é seguro e desejável paralelizar o laço afetado. Se a pessoa se equivocou na sua análise, há uma grande probabilidade da aplicação resultante produzir resultados incorretos.

O programador identifica o paralelismo sem restringir o modo como o compilador vai explorar esse paralelismo. Isso permite o transporte do código OpenACC para outros tipos de dispositivos, além do dispositivo para o qual o código foi primeiramente desenvolvido, porque detalhes específicos sobre como paralelizar o código são deixados para o compilador decidir, ao invés de isso ser especificado explicitamente no código-fonte.

Ao se tentar paralelizar múltiplos laços dentro da mesma região paralela podem ocorrer problemas de desempenho ou resultados inesperados. No próximo exemplo, o compilador tem completa liberdade para mapear o primeiro laço para algumas gangues e o segundo laço para as gangues restantes. Isso pode levar a uma condição de corrida que, em alguns sistemas apenas (e esse é o problema), pode resultar em uma computação incorreta. Logo, **não** se deve utilizar esta construção.

```
#pragma acc parallel
#pragma acc loop
    for (i = 0; i < 100; i ++) {
         a [i] = b[i];
    }
#pragma acc loop
    for (i = 0; i < 100; i ++) {
         c [i] = b[i]* a[i]*2;
    }
```

A forma adequada de paralelização é a utilização de várias diretivas **parallel loop**, uma para cada laço, evitando assim as condições de corrida descritas acima.

```
#pragma acc parallel loop
    for (int i = 0 ; i < N; i++)
        a[i] = 0;
#pragma acc parallel loop
    for (int j = 0 ; j < M; j++)
         b[j] = 0;
```

7.3 Movimentação de dados

Com dito anteriormente, um grande fator de impacto no desempenho do processamento paralelo é a movimentação de dados, principalmente quando se faz o processamento dos dados em lugares diferentes. Quando se usa processamento em aceleradores nem sempre é possível carregar todos os dados para o acelerador, isso ocorre em geral porque a memória da CPU tem uma capacidade maior do que a dos aceleradores, embora os aceleradores tenham maior largura de banda (Figura 7.4).

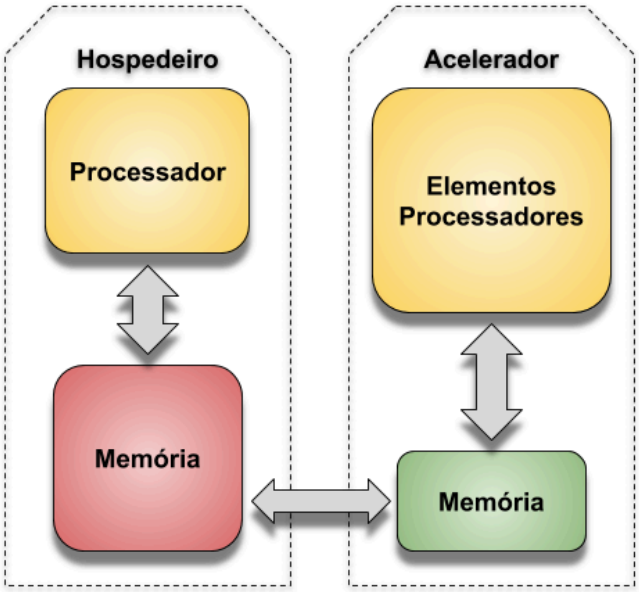

Figura 7.4: Modelo básico de movimentação de dados entre hospedeiro e o acelerador

A movimentação de dados entre o hospedeiro e o acelerador é feita através de um barramento de E/S, que é lento em comparação com a largura de banda de memória. Por sua vez, o acelerador não é capaz de executar o processamento dos dados enquanto eles não estejam na sua memória local.

Para realizar a movimentação de dados entre o hospedeiro e o acelerador durante a execução do programa é necessário o uso das cláusulas de dados. As cláusulas de dados dão ao programador controle adicional sobre como e quando os dados são criados e copiados de/para um dispositivo. As cláusulas de movimentação de dados podem ser usadas nas construções **data**, **kernels** ou **parallel** e têm como principais caraterísticas:

- Definir a região do código na qual os dados permanecem no acelerador;
- Definir quais dados são compartilhados entre todos os *kernels* de uma região paralela;
- Realizar transferências de dados explícitas.

Se não houver uma cláusula **default(none)** (Seção 7.4.2) para aquela região, o compilador vai determinar implicitamente os atributos para as variáveis que não estiverem listadas na diretiva **data**. Por exemplo, as variáveis escalares receberão um atributo **firstprivate**; os índices dos laços serão **private** por padrão; vetores e matrizes serão compartilhados por padrão, sendo que o compilador escolhe o tipo: **copyin, copyout** etc. Maiores detalhes serão vistos na próxima seção.

7.3.1 Diretiva data

A diretiva **data** define quais escalares, arranjos e subarranjos devem ser alocados na memória do dispositivo durante o tempo de execução de uma região, definindo também se os dados devem ser copiados da memória do hospedeiro para a memória do dispositivo na entrada da região e/ou copiados da memória do dispositivo para a memória do hospedeiro, na saída da região. A sintaxe da diretiva **data** é vista a seguir:

#pragma acc data [cláusula]

De uma forma mais geral, as cláusulas da diretiva **data** são utilizadas para definir o escopo das variáveis compartilhadas pelas *threads* e gangues que executam no acelerador. Uma descrição mais detalhada sobre a aplicação de cada uma dessas cláusulas pode ser vista a seguir:

- **copy** — Cria espaço para as variáveis listadas no dispositivo, inicia as variáveis copiando dados para o dispositivo no início da região, copia os resultados de volta para o hospedeiro no final da região e finalmente libera o espaço no dispositivo quando terminar.

- **copyin** — Cria espaço para as variáveis listadas no dispositivo, inicia a variável copiando os dados para o dispositivo no início da região e libera o espaço no dispositivo quando terminar, sem copiar os dados de volta para o hospedeiro.

- **copyout** — Cria espaço para as variáveis listadas no dispositivo, mas não as inicia. No final da região, copia os resultados de volta para o hospedeiro e libera o espaço no dispositivo.

- **create** — Cria espaço para as variáveis listadas e as libera no final da região, mas não copia nenhum dos dados de/para o dispositivo.

- **present** — As variáveis listadas já estão presentes no dispositivo, portanto, nenhuma outra ação precisa ser

executada. Isso é usado com mais frequência quando existe uma região de dados em uma rotina de maior nível que já movimentou os dados para o dispositivo.

- **deviceptr** — As variáveis listadas usam a memória do dispositivo que foi gerenciada fora do OpenACC, portanto as variáveis devem ser usadas no dispositivo sem qualquer conversão de endereço. Esta cláusula é geralmente usada quando o OpenACC é misturado com outro modelo de programação, como CUDA.

O exemplo seguinte apresenta uma movimentação simples de dados.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main( int argc, char* argv[]) { /* acc_data.c  */
int n = 30000000;                   /* tamanho dos vetores  */
double *restrict a, *restrict b;    /* vetores de entrada */
double *restrict c;                 /* vetor de saída */
size_t bytes = n*sizeof(double);    /* tamanho em bytes de cada
vetor */
    /*  Alocação de memória para cada vetor */
    a = (double*)malloc(bytes);
    b = (double*)malloc(bytes);
    c = (double*)malloc(bytes);
    /* Atribui os valores iniciais em paralelo */
#pragma acc parallel loop gang copyout(a[0:n], b[0:n] )
    for (int i = 0; i < n; i++) {
         a[i] = sin(i)*sin(i);
         b[i] = cos(i)*cos(i);
    }
    double soma = 0.0;
      /* Executa as iterações em paralelo e faz a redução ao
final */
     /* O vetor 'c' só existe no acelerador, a e b são copiados
do host para o acelerador */
#pragma acc parallel loop gang reduction(+:soma) create(c[0:n])
copyin(a[0:n], b[0:n])
    for(int i = 0; i < n; i++) {
```

```
            c[i] = a[i] + b[i];
            soma += c[i];
        }
        soma = soma/(double)n;
        /* Imprime o resultado */
        printf("Resultado final: %f\n", soma);
        /* Libera a memória alocada */
        free(a);
        free(b);
        free(c);
        return 0;
}
```

Nesse exemplo, no primeiro laço, depois da atribuição de valores iniciais aos vetores Math input error e Math input error, os vetores são copiados, ao final do laço, da memória do dispositivo para a memória do hospedeiro, conforme determinado pela cláusula **copyout (a[0:n], b[0:n])**.

Em seguida, no segundo laço, os vetores Math input error e Math input error são copiados, no início do laço, da memória do hospedeiro para a memória do dispositivo, conforme determinado pela cláusula **copyin (a[0:n], b[0:n])**. Já o vetor Math input error, de uso temporário, será criado apenas na memória do dispositivo, sem ser copiado em nenhum sentido, nem no início nem ao final do laço, conforme determinado pela cláusula **create (c[0:n])**.

A operação de redução utilizada no exemplo serve para agregar os valores parciais calculados por cada gangue em único valor ao final do laço. Maiores detalhes serão apresentados na Seção 7.4.4. Note que a movimentação de dados mostrada nesse exemplo ainda pode ser melhorada, mas isso é deixado como exercício para o leitor.

7.3.2 Memória Unificada (Unified Memory)

As memórias da CPU e GPU são normalmente separadas, ou seja, as informações armazenadas em seus sistemas de memórias não são

compartilhadas. A movimentação de dados entre os dois sistemas de memórias é realizada sempre que existe a necessidade de processamento de alguma informação.

Uma forma de solucionar este problema é o uso de memória unificada. A memória unificada é um espaço de endereçamento único acessível tanto pela CPU como pela GPU. Com uso dessa tecnologia de *hardware/software*, as aplicações alocam dados que podem ser lidos e escritos tanto pelas CPUs como pelas GPUS (Figura 7.5).

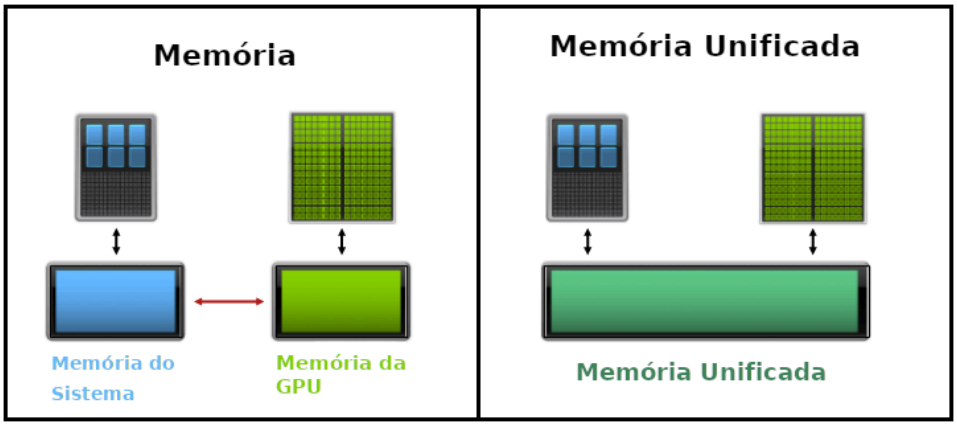

Figura 7.5: Modelo de memória tradicional e memória unificada

Quando o código em execução em uma CPU ou GPU acessa dados alocados dessa maneira (*CUDA managed memory*), o sistema de *software* CUDA e/ou *hardware* se encarrega de migrar as páginas para a memória do processador que estiver acessando esses dados.

Destacamos que a arquitetura de GPU Pascal é a primeira com suporte de *hardware* para falha e migração de página de memória virtual, por meio de seu mecanismo de migração de página. As GPUs mais antigas baseadas nas arquiteturas Kepler e Maxwell também suportam uma forma mais limitada de memória unificada.

A alocação de memória unificada é realizada explicitamente em CUDA substituindo-se as chamadas para a função **malloc()** por

chamadas à função **cudaMallocManaged()**, uma função de alocação de memória que retorna um ponteiro acessível por qualquer processador.

Já no OpenACC essa substituição das chamadas é feita automaticamente pelo compilador. Para o uso da Memória Unificada no OpenACC com o compilador PGI, basta utilizar usar a opção *-ta=tesla:managed* na compilação do código do programa. Portanto, as cláusulas e diretivas de dados OpenACC não são necessárias para dados "gerenciados". Elas são essencialmente ignoradas e, de fato, podem ser omitidas.

O uso de memória gerenciada se aplica apenas aos dados alocados dinamicamente. Dados estáticos (variáveis externas e estáticas em C, módulos em Fortran, blocos comuns e variáveis salvas) e dados locais das funções são ainda manipulados pelo ambiente de execução do OpenACC.

Embora o uso da memória unificada ofereça uma grande simplificação da programação, o desempenho final é variável em função de cada aplicação e deve ser avaliado com cuidado antes de sua utilização em produção.

7.4 Cláusulas das diretivas parallel

Algumas cláusulas cuja utilização é permitida com as diretivas **parallel** serão apresentadas a seguir. Algumas delas, eventualmente, são utilizadas também com a diretiva **kernels**. Quando for esse o caso, isso será claramente indicado. Essas cláusulas servem para indicar, por exemplo, qual deve ser o escopo das variáveis; como vai ser feita a movimentação entre o hospedeiro e o acelerador; se vai ser realizada uma operação de redução ao final da região; ou o número de gangues, trabalhadores e comprimento dos vetores utilizados para executar os laços na região *parallel*.

Destacamos que todas as cláusulas utilizadas na diretiva **data** são utilizadas também com as diretivas **parallel** e **kernels**.

7.4.1 Cláusulas private e firstprivate

A cláusula **private** da construção **parallel** vai privatizar as variáveis listadas para cada gangue na região paralela. Cada gangue vai ter uma cópia própria das variáveis da lista. Vemos a seguir seu formato:

#pragma acc parallel private (lista-de-variáveis)

A cláusula **firstprivate** da construção **parallel** vai privatizar as variáveis listadas para cada gangue na região paralela, e a cópia será iniciada com o valor desse item na *thread* do hospedeiro quando a construção **parallel** for encontrada. A sua sintaxe é a seguinte:

#pragma acc parallel firstprivate (lista-de-variáveis)

Na realidade, no OpenACC, escalares são definidos como **firstprivate** por padrão, então não há necessidade de utilização da cláusula **private**. O seu uso é necessário apenas com arranjos ou quando um escalar é passado por referência para uma rotina no dispositivo. Apresentamos mais detalhes no exemplo:

```
#include <stdio.h>
int main(int argc, char *argv[]) { /* acc_private.c  */
int a[2]={10,20}, b[2]={100,200}, c=1000;
#pragma acc parallel private(a) firstprivate(b) num_gangs(4)
    {
        a[0] += 1;
        b[0] += 1;
        c += 1;
            printf("Valor de a = %d, b = %d, c = %d \n",
a[0],b[0],c);
    }
```

```
    return 0;
}
```

Uma possível saída para esse programa é listada a seguir, sendo que alguns valores podem variar de uma implementação para outra.

```
Valor de a = 1, b = 101, c = 1001
Valor de a = 1, b = 101, c = 1001
Valor de a = 1, b = 101, c = 1001
Valor de a = 1, b = 101, c = 1001
```

O vetor *a*, declarado como privado, não tem o seu valor inicial copiado para dentro da região paralela. O valor que o vetor vai receber dentro de cada gangue é dependente de implementação. De forma contrária, o vetor *b*, declarado como *firstprivate*, receberá valor inicial 100 em cada gangue. Já a variável escalar *c*, apesar de não ter nenhuma declaração explícita, é considerada como *firstprivate* pelo compilador, e tem seu valor inicial, 1000, copiado para cada uma das gangues.

A diretiva **loop** também faz uso da cláusula **private**, cujos detalhes são apresentados na Seção [7.5.6].

7.4.2 Cláusula default(none)

A cláusula **default(none)** é opcional e serve para indicar que o compilador não deve determinar implicitamente o atributo de dados das variáveis, mas sim obrigar que todas as variáveis ou arranjos utilizados na região, cujos atributos de dados não estejam predeterminados, apareçam explicitamente em uma cláusula **data** para a região ou para uma região na qual essa região esteja contida.

#pragma acc parallel default (none)

Conforme já dito, se não houver uma cláusula **default(none)** o compilador vai determinar implicitamente quais serão os atributos das variáveis que não constarem na cláusula **data**. Por exemplo, as

variáveis escalares receberão um atributo *firstprivate*; os índices dos laços serão *private* por padrão; sendo que vetores e matrizes terão a forma de compartilhamento escolhida pelo compilador: copyin, copyout etc.

7.4.3 Cláusula independent

Em uma diretiva **kernels**, a cláusula **independent** indica para o compilador que as iterações do laço são independentes entre si. Isso permite a geração, pelo compilador, de código para executar as iterações do laço em paralelo sem nenhuma sincronização. Em uma diretiva **parallel**, o uso da cláusula **independent** é implícito em todas as diretivas **loop** sem a cláusula **seq**.

É um erro de programação indicar o uso da cláusula **independent** em um laço dentro de uma região *kernels* se alguma iteração escreve em uma variável que seja lida ou escrita por outra iteração, exceto para as variáveis na cláusula **reduction**.

7.4.4 Cláusula reduction

A cláusula **reduction** funciona de forma similar à cláusula **private**, sendo gerada uma cópia privada das variáveis, mas existe uma redução ao final da região paralela de todas as cópias privadas em um único resultado final, cujo valor é retornado ao sair da região paralela. A redução deve ser especificada apenas em uma variável **escalar** e as operações de redução possíveis são: soma; multiplicação; máximo; mínimo; Math input error bit a bit; Math input error bit a bit; Math input error; Math input error lógico; Math input error lógico, conforme visto na Tabela 7.1. O formato para a cláusula **reduction** é visto a seguir:

#pragma acc parallel reduction (operador:variável)

Tabela 7.1: Operadores da cláusula reduction

operador	operação	valor inicial
+	soma	0
*	multiplicação	1
max	máximo	menor rep.
min	mínimo	maior rep.
&	"e" bit a bit	~0
|	"ou" bit a bit	0
^	"ou exclusivo" bit a bit	0
&&	"e" lógico	1
||	"ou" lógico	0

Para as reduções com **max** e **min** os valores iniciais são o menor valor representável e o maior valor representável para o tipo de dados da variável de redução, respectivamente. Os tipos de dados suportados nessa operação são os tipos de dados numéricos em C e C++ (int, float, double e complex).

A diretiva **loop** também faz uso da cláusula **reduction**, com os mesmos operadores, cujos detalhes e exemplos podem ser vistos na Seção 7.5.7.

7.4.5 Cláusula if

A cláusula **if** é opcional nas construções **parallel** e **kernels**. Quando não houver nenhuma cláusula **if**, o compilador vai gerar código para executar a região no dispositivo acelerador. Quando houver uma cláusula **if**, o compilador vai gerar duas cópias da região paralela, uma cópia para executar a região no acelerador e uma cópia para executar na *thread* local no hospedeiro. Quando a condição avaliada for diferente de 0, a cópia do acelerador será executada. Em caso contrário, a cópia da região para a *thread* local será executada.

#pragma acc parallel if (condição)

7.4.6 Cláusulas num_gangs, num_workers e vector_length

O comportamento padrão é deixar ao compilador determinar quantos trabalhadores e o comprimento dos seus respectivos vetores nos diversos laços em uma região *parallel*. O número de gangues é definido pelo ambiente de execução quando o programa for executado, sendo que o uso da memória é normalmente o critério limitante.

Contudo, o programador pode definir esses parâmetros dentro de uma região **parallel** ou **kernels** com o uso das cláusulas **num_gangs(N), num_workers(M), vector_length(Q)**.

Essas cláusulas são úteis principalmente se o código utilizar uma estrutura de dados de difícil análise para o compilador. O número ótimo de gangues é altamente dependente do tipo de arquitetura, portanto use-as com cautela. Vejamos então:

num_gangs

Na cláusula **num_gangues(N)**, o valor da expressão inteira **N** define o número de gangues paralelas que vão executar a região. Se essa cláusula não for especificada, um valor padrão dependente da implementação é utilizado. É equivalente ao parâmetro *gridDim* em CUDA.

num_workers

Na cláusula **num_workers(M)**, o valor da expressão inteira **M** define o número de trabalhadores que serão ativados em cada gangue. Se essa cláusula não for especificada, um valor padrão dependente da implementação é utilizado, podendo ser 1 ou também haver um valor diferente para cada construção **parallel**. É equivalente ao parâmetro *blockDim* em CUDA.

vector_length

Na cláusula **vector_length(Q)**, o valor da expressão inteira **Q** define o tamanho de cada vetor. Essa cláusula determina o comprimento do vetor para ser utilizado em operações vetoriais ou SIMD. Se essa cláusula não for especificada, um valor padrão dependente da implementação é utilizado. Esse comprimento de vetor será utilizado para todos os laços anotados com a cláusula **vector** em uma diretiva **loop**, assim como nos laços automaticamente vetorizados pelo compilador. Pode haver limites definidos pela implementação para os valores de comprimento de vetor utilizados na expressão inteira.

#pragma acc parallel num_gangs(expr-inteira)
#pragma acc parallel num_workers(expr-inteira)
#pragma acc parallel vector_length(expr-inteira)

Observações quanto ao particionamento dos laços

Usualmente não se utiliza cláusula **num_workers**, sendo seu uso limitado ao caso em que o tamanho do vetor é muito pequeno, quando o uso cláusula **worker** pode aumentar o paralelismo disponível na gangue. Para as demais situações, o comum é definir um número de gangues (ou blocos de *threads*) e o comprimento do vetor (ou *threads*). Assim sendo, o comprimento do vetor (número de *threads*) entre 256 e 512 é usualmente um bom número, já que blocos muito pequenos limitam o número de *threads* concorrentes devido à limitação no número máximo de blocos/SM concorrentes. Um bloco com um número muito grande de *threads* pode ter um alto custo de sincronização/barreira entre as *threads* do bloco, impactando o desempenho. Em todo caso, como já dito anteriormente, o tamanho máximo de uma gangue deve ser de 1024 *threads*.

Para os dispositivos NVIDIA disponíveis até a data, é preferível a utilização de um número de *threads* por bloco que seja múltiplo de

32, se possível. Isso porque o tamanho de um *warp* nas GPUs atuais da NVIDIA é igual a 32 *threads* e facilita o acesso a endereços adjacentes de memória compartilhada, que é feito de uma forma agregada para reduzir o número de transações.

O mapeamento mais óbvio é ter o número de gangues vezes o número de trabalhadores vezes o número de vetores igual ao tamanho do problema. Contudo, é interessante ajustar esses valores para que cada *thread* possa realizar múltiplas partes de trabalho, amortizando o custo de preparação para *kernels* simples. Isso implica que se deve ter um número de *threads* suficiente para ocupar a GPU diversas vezes (10 ou mais), o que dá pelo menos algumas dezenas de milhares de *threads*.

7.5 Cláusulas da diretiva loop

A seguir apresentamos algumas cláusulas da diretiva **loop** que são utilizadas com diversos objetivos, tais como definir o escopo das variáveis; se vai ser realizada uma operação de redução; ou ainda como a execução do laço vai ser dividida entre gangues, trabalhadores e vetores.

7.5.1 Cláusula gang

A cláusula **gang** em uma região *parallel* especifica que as iterações dos laços associados devem ser executadas em paralelo através da distribuição das iterações entre as gangues criadas pela diretiva **parallel**. O número de gangues é controlado pela diretiva **parallel**, sendo que apenas o argumento **static**, visto a seguir, é permitido.

#pragma acc loop gang [(expr-inteira)] [(static:[*] [expr-inteira])

Quando utilizada em uma construção **kernels**, a cláusula **gang** especifica que as iterações do laço ou laços associados devem ser

executadas em paralelo através das gangues. Um argumento, sem nenhuma palavra-chave ou com a palavra-chave **num**, só é permitido se a cláusula **num_gangs não** aparecer na construção **kernels**. Nesse caso, ele especifica quantas gangues devem ser utilizadas para executar as iterações do laço.

As iterações do laço não devem ter dependências de dados, a não ser para as variáveis especificadas na cláusula **reduction**, como veremos na Seção 7.5.7. Um laço com a cláusula **gang** não pode conter outro laço com outra cláusula **gang**, a menos que seja uma região *kernels* ou *parallel* aninhada.

O escalonamento de iterações do laço para as gangues não é especificado a menos que o argumento **static** seja utilizado. Nesse caso, o valor inteiro informado após o argumento é o tamanho do *chunksize*, ou seja, a quantidade de iterações atribuídas a cada gangue, começando da gangue 0 e prosseguindo com as demais em ordem. Se for utilizado um asterisco (*) a implementação vai escolher o tamanho do *chunksize*.

7.5.2 Cláusula worker

Em uma região *parallel*, a cláusula **worker** determina que as iterações do laço ou laços associados devem ser executadas em paralelo, distribuindo as iterações entre os vários trabalhadores dentro de uma única gangue.

Uma diretiva **loop** com uma cláusula **worker** faz com que uma gangue mude do modo trabalhador único para o modo trabalhador particionado. Em contraste com a cláusula **gang**, a cláusula **worker** primeiro ativa o paralelismo adicional no nível do trabalhador e, em seguida, distribui as iterações do laço entre esses trabalhadores. Ao contrário do uso em uma região *kernels*, nenhum argumento é permitido para esta cláusula em uma região *parallel*. As iterações do laço não devem ter dependências de dados, exceto para as variáveis especificadas em uma cláusula de redução.

Em uma região *kernels* do acelerador, um argumento numérico só é permitido caso a cláusula **num_workers não** apareça na diretiva **kernels**. Desse modo, se um argumento numérico for especificado, com ou sem o qualificador **num**, ele indica quantos trabalhadores por gangue devem ser usados para executar as iterações desse laço.

A região de um laço com a cláusula **worker** não pode conter um laço com uma cláusula **gang** ou **worker**, a menos que seja dentro de uma região *kernels* ou *parallel* aninhada. Todos os trabalhadores concluirão a execução de todas as iterações que lhe forem atribuídas, antes que qualquer trabalhador prossiga além do final do laço.

#pragma acc loop worker[([num:]expr-inteira)]

7.5.3 Cláusula vector

Em uma região *parallel* do acelerador, a cláusula **vector** especifica que as iterações do laço ou laços associados devem ser executadas no modo vetorial ou SIMD. Uma construção de laço com uma cláusula **vector** faz com que um trabalhador faça a transição do modo de vetor único para o modo de particionamento de vetor. Semelhante à cláusula **worker**, a cláusula **vector** primeiro ativa o paralelismo adicional no nível de vetor e, em seguida, distribui as iterações do laço entre esses vetores. As operações serão executadas usando vetores de comprimento especificado ou escolhido para a região paralela. A sua sintaxe é a seguinte:

#pragma acc loop vector [([length:]expr-inteira)]

Em uma região *kernels* do acelerador, um argumento numérico só é permitido se a cláusula **vector_length** não aparecer na diretiva **kernels**. Nesse caso, se um argumento for especificado, as iterações serão processadas por vetores com esse comprimento; se nenhum argumento for especificado, a implementação escolherá um comprimento de vetor apropriado.

A região de um laço com a cláusula **vector** não pode conter um laço com uma cláusula **gang**, **worker** ou **vector** a menos que dentro de uma região *parallel* ou *kernels* aninhada. Todos os vetores vão completar a execução das iterações que lhe foram atribuídas antes que qualquer vetor possa prosseguir além do final do laço.

7.5.4 Cláusula seq

A cláusula **seq** especifica que o laço ou laços associados devem ser executados sequencialmente pelo acelerador. Essa cláusula vai se sobrepor a qualquer paralelização ou vetorização automáticas que poderiam ser feitas pelo compilador.

7.5.5 Cláusula auto

A cláusula **auto** determina que a implementação deve selecionar se vai ser aplicado paralelismo de gangue, trabalhador ou vetor ao laço, devendo ainda analisar o laço para identificar se as iterações do laço são independentes de dados. Em uma construção **kernels**, um laço sem nenhuma cláusula **gang**, **worker**, **vector** ou **seq** é tratado como se tivesse uma cláusula **auto**. O exemplo seguinte apresenta o uso das cláusulas mencionadas nesta seção.

```
#include <stdlib.h>
void saxpy(int n, float a, float * restrict x, float * restrict
y) {
    /* Divide as iterações entre as gangues */
    #pragma acc parallel num_gangs(10)
    #pragma acc loop gang (static:10000)
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
int main(int argc, char *argv[]) { /* acc_gang.c  */
int n = 1<<24;                     /* 16 milhões de floats */
float *x = (float*)malloc(n*sizeof(float));
float *y = (float*)malloc(n*sizeof(float));
    /* Define o número de trabalhadores e o tamanho do vetor */
```

```
    #pragma acc parallel num_workers(8) vector_length(128)
     /* Divide as iterações entre as gangues, workers e vetores
*/
    #pragma acc loop gang worker vector
    for (int i = 0; i < n; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(n, 3.0f, x, y);
    free(x);
    free(y);
    return(0);
}
```

7.5.6 Cláusula private

A cláusula **private** define que cada iteração do laço terá sua própria cópia das variáveis listadas. Variáveis referenciadas em um laço, que não estejam em uma cláusula **private** e não sejam privadas por definição, não serão privatizadas para nenhuma *thread* que executar as iterações do laço.

Por exemplo, se cada laço contiver uma matriz pequena e temporária denominada *tmp* usada durante seu cálculo, essa variável deverá ser tornada privada para cada iteração do laço, a fim de garantir resultados corretos. Se a matriz *tmp* não for declarada privada, *threads* executando diferentes iterações do laço podem acessar a matriz *tmp* compartilhada de modo imprevisível, resultando em uma condição de corrida e em resultados potencialmente incorretos. Apresentamos a seguir a sintaxe da cláusula privada.

#pragma acc loop private (lista-de-variáveis)

Há alguns casos especiais que devem entendidos sobre variáveis escalares em laços. Primeiro, os iteradores do laço são privados por padrão, então eles não precisam ser listados na diretiva **private**. Segundo, a menos que seja especificado o contrário, qualquer

variável escalar acessada dentro de um laço com a diretiva combinada **parallel loop** será definida como **firstprivate** por padrão, ou seja, uma cópia será feita para cada iteração do laço e terá um valor inicial igual ao da variável escalar ao se entrar na região.

Finalmente, quaisquer variáveis (escalares ou não) declaradas dentro de um laço em C ou C++ serão feitas privadas para as iterações do laço por padrão. Essas diferenças fazem o uso da diretiva **private** muito mais raro no OpenACC comparado ao OpenMP.

Nota: Veja na Seção 7.4 que a construção **parallel** também tem cláusulas **private** e **firstprivate** que privatizam a lista de variáveis para cada gangue na região paralela.

7.5.7 Cláusula reduction

A diretiva **loop** possui uma cláusula com o mesmo nome **reduction** e com funcionamento similar à diretiva **parallel**. Uma cópia privada da variável é gerada para cada iteração do laço, mas é feita uma redução em um único resultado final, cujo valor é retornado ao final da região paralela ou *kernels*.

Em uma região paralela, se a cláusula de redução for utilizada em um laço com cláusulas **vector** ou **worker** (e sem a cláusula **gang**), e a variável escalar também aparecer em uma cláusula **private** na construção **parallel**, o valor da cópia privada da variável escalar será atualizado na saída do laço. Se a variável escalar não aparecer na lista da cláusula **private** na construção **parallel**, ou se a cláusula **gang** tiver sido utilizada no laço, o valor da variável escalar só será atualizado no final da região paralela.

O formato da cláusula **reduction** é mostrado a seguir, onde *operador* deve ser substituído com a operação desejada e *variavel*

deve ser substituída com a variável sendo reduzida, conforme visto na Tabela 7.1. O exemplo adiante mostra a sua utilização.

#pragma acc loop reduction(operador:variavel)

```
#include <stdio.h>
#define SOMA_INICIAL 1000
int main(int argc, char *argv[]) { /* acc_reduction.c  */
int i, n = 25;
int soma, a[n];
int ref = SOMA_INICIAL + ((n-1)*n)/2;
    for (i = 0; i < n; i++)
         a[i] = i;
    soma = SOMA_INICIAL;
       printf("Valor da soma antes da região paralela: %d
\n",soma);
    #pragma acc parallel loop reduction(+:soma)
    for (i = 0; i < n; i++)
         soma += a[i];
    /*-- Fim da redução paralela --*/
         printf("Valor da soma depois da região paralela:
%d\n",soma);
     printf("Verificação do resultado: soma = %d (deveria ser
%d)\n",soma,ref);
    return(0);
}
```

7.5.8 Cláusula collapse

No OpenACC, se nenhuma cláusula **collapse** estiver presente, apenas o laço imediatamente a seguir será associado com a diretiva **loop**, ou seja, uma diretiva **loop** é necessária para cada laço aninhado. Isso tende a ser complicado, especialmente se vários laços devem ser tratados da mesma maneira. A cláusula **collapse** é útil nesse caso. O argumento para a cláusula **collapse** é um número inteiro positivo constante, especificando quantos laços fortemente aninhados serão associados para a criar um novo laço.

É dependente de implementação se uma cláusula **gang**, **worker** ou **vector** na diretiva **loop** será aplicada a cada laço ou ao espaço de iteração linearizado.

Quais as vantagens em usar a cláusula **collapse**?

- colapsar os laços externos para permitir a criação de mais gangues;
- colapsar os laços internos para permitir comprimentos de vetor mais longos;
- colapsar todos os laços, quando for possível, para fazer as duas coisas: ter mais gangues criadas e vetores maiores.

Essa cláusula é especialmente útil quando alguns laços não tem um número total de iterações suficientemente grande para fazer uso efetivo do acelerador. A sua sintaxe é vista a seguir:

#pragma acc loop collapse(n)

Apresentamos no próximo exemplo um trecho de código com um laço com o uso dessa cláusula, seguido de um laço exemplificando o efeito do seu uso.

```
#pragma acc parallel loop collapse(2)
    for (int i = 0; i < N; i++)
         for (int j = 0; j < M; j++)

#pragma acc parallel loop
    for (int ij = 0; ij < N*M; ij++)...
```

7.5.9 Cláusula tile

A adição da cláusula **tile** à diretiva **acc loop** é um novo recurso disponível no OpenACC 2.0. Com a cláusula **tile** é possível otimizar a execução do laço através da operação em blocos menores para

explorar melhor o acesso aos dados. Considere o seguinte exemplo de transposição de matriz.

```
#pragma acc parallel loop private(i,j) tile(8,8)
    for (i = 0; i < rows; i++)
         for (j = 0; j < cols; j++)
              out[i*rows + j] = in[j*cols + i];
```

Ao adicionar a cláusula tile (8,8) ao laço paralelo, serão criados automaticamente pelo compilador dois laços adicionais que funcionam em um bloco 8x8 (*tile*) da matriz antes de passar para o próximo bloco. Com isso o compilador faz a otimização dentro do bloco, com o objetivo de obter melhor desempenho.

Embora uma transposição de matriz não tenha muita reutilização de dados, outros algoritmos podem experimentar uma melhoria significativa no seu desempenho, explorando a localidade e a reutilização de dados nos laços disponíveis. A sintaxe da cláusula é a seguinte:

#pragma acc loop tile(lista-expr-tam)

onde *(lista-expr-tam)* é do tipo (N[,M,...]). O primeiro argumento na *lista-expr-tam* corresponde ao laço mais interno dos Math input error laços associados, e o último elemento corresponde ao laço associado mais externo. Se o tamanho for especificado com um asterisco, a implementação escolherá um valor apropriado.

7.5.10 Cláusula device_type

Com o OpenACC 1.0, diretivas de pré-processador eram necessárias para ajustar as diretivas para uso em aceleradores específicos. Além de dificultar a manutenção do código, devido à duplicação de diretivas, isso tornava impossível o suporte a vários tipos de dispositivos no mesmo programa executável. A versão 2.0 do OpenACC permite que determinadas cláusulas sejam fornecidas

especificamente para determinadas arquiteturas, sendo sua resolução feita em tempo de execução.

As diretivas do OpenACC podem especificar cláusulas ou argumentos de cláusulas diferentes para dispositivos diferentes usando a cláusula **device_type**. Uma única diretiva pode ter uma ou várias cláusulas **device_type**. As cláusulas relacionadas após um tipo de dispositivo são específicas para aquele tipo de dispositivo. Cláusulas associadas a um **device_type** com um asterisco como argumento se aplicam a todos os tipos de dispositivo que não estejam relacionados em nenhuma cláusula **device_type**.

Apenas as cláusulas **async**, **wait**, **num_gangs**, **num_workers** e **vector_length** podem aparecer após uma cláusula **device_type** de uma diretiva **kernels** ou **parallel**. Apenas as cláusulas **collapse**, **gang**, **worker**, **vector**, **seq**, **auto** e **tile** podem aparecer após uma cláusula **device_type** de uma diretiva **loop**.

Os tipos de dispositivos suportados são dependentes de implementação. Dependendo da implementação e do ambiente de compilação, uma implementação pode suportar apenas um tipo de dispositivo ou vários tipos de dispositivo, mas apenas um por vez; ou vários tipos de dispositivos em uma única compilação.

#pragma acc loop device_type(lista-tipo-dispositivo)

A cláusula **vector_length** quando utilizada em associação com a cláusula **device_type** especifica o tamanho do vetor que será usado por cada dispositivo em um laço paralelo (com a diretiva **parallel loop**).

A seguir listamos um trecho de código onde é especificado um comprimento diferente de vetor, dependendo do tipo de acelerador utilizado.

```
#pragma acc parallel loop \
        device_type(nvidia) vector_length(256) \
```

```
    device_type(radeon) vector_length(512) \
                        vector_length(64)
for (int i = 0; i < n; ++i) {
     c[i] = a[i] + b[i];
}
```

Nesse exemplo, se o laço for utilizar um acelerador da **NVIDIA**, o compilador utilizará um comprimento vetorial de 256; se for um acelerador **Radeon**, o comprimento de vetor será de 512; e para qualquer outro acelerador diferente será usado um comprimento vetorial de 64. As duas cláusulas podem ser utilizadas em conjunto com as demais cláusulas do OpenACC.

7.6 Diretivas avançadas

São apresentadas nesta seção algumas diretivas especiais para extração de paralelismo em rotinas ou para evitar condições de corrida.

7.6.1 Diretiva routine

As chamadas de função ou subrotina em laços paralelos podem ser problemáticas para os compiladores, pois nem sempre é possível para o compilador ver todos os laços de uma só vez. Os compiladores OpenACC 1.0 eram forçados a fazer *inline* de todas as rotinas chamadas em regiões paralelas ou não paralelizar laços contendo chamadas de rotina.

O OpenACC 2.0 introduziu a diretiva **routine**, que instrui o compilador a criar uma versão de dispositivo da função ou subrotina para que possa ser chamada de uma região de dispositivo. Para leitores já familiarizados com a programação CUDA, essa funcionalidade é semelhante ao especificador da função __*device*__. A sua sintaxe é a seguinte.

#pragma acc routine [cláusulas]

Para orientar a otimização, as cláusulas servem para informar ao compilador se a rotina deve ser criada para paralelismo no nível de **gang**, **work**, **vector** ou **seq**. Várias cláusulas podem ser especificadas para as rotinas, que serão então chamadas com vários níveis de paralelismo. Fazer isso corretamente exige a colocação de uma diretiva **routine** antes da definição da rotina para chamar a rotina com o nível certo de paralelismo.

```
#pragma acc routine vector
void foo(float* v, int i, int n) {
#pragma acc loop vector
    for (int j = 0; j < n; ++j)
         v[i*n+j] = 1.0f/(i*j);
}

#pragma acc parallel loop
    for (int i = 0; i < n; ++i)
         foo(v,i);  //chamada no dispositivo
```

Quando a rotina *foo* é chamada a partir do código do hospedeiro, ele será executado no hospedeiro, incrementando as variáveis do hospedeiro. Quando chamado de dentro de uma construção paralela do OpenACC, as variáveis do dispositivo serão incrementadas.

Teoricamente, essa diretiva permitiria o uso de funções recursivas, contudo há alguns fatores que limitam a profundidade da recursão. Por exemplo, os dispositivos NVIDIA estão limitados a 16 níveis de recursão, assim como os dispositivos AMD possuem seus próprios limites.

Nota: a partir da versão 14.9 do compilador PGI, uma diretiva **routine** sem nenhuma cláusula de nível de paralelismo (**gang**, **worker** ou **vector**) será tratada como se uma cláusula **seq** estivesse presente.

7.6.2 Diretiva atomic

Quando uma ou mais iterações de um laço precisam acessar ao mesmo tempo um elemento na memória, condições de corrida podem ocorrer. Por exemplo, se uma iteração do laço está modificando o valor contido em uma variável e outra iteração está tentando ler a mesma variável em paralelo, pode-se obter resultados diferentes dependendo de qual iteração ocorreu primeiro.

Em programas seriais, os laços sequenciais garantem que a variável será modificada e lida em uma ordem previsível, mas os programas paralelos não asseguram que uma iteração específica de um laço ocorrerá antes da outra. Em casos de algoritmos simples, como encontrar uma soma, valor máximo ou mínimo, uma operação de redução garante a execução correta do programa.

Para operações mais complexas, a diretiva **atomic** certifica que não haverá duas *threads* (**gang**, **work** ou **vector**) executando a operação nela contida simultaneamente. O uso da diretiva **atomic** é às vezes uma parte necessária do processo de paralelização para garantir a correção do código.

A diretiva **atomic** aceita uma das quatro cláusulas seguintes para declarar o tipo de operação contida na região:

- A operação **read** assegura que duas iterações de um laço não farão leituras da região ao mesmo tempo;
- A operação **write** garantirá que não haja duas iterações realizando escrita na região ao mesmo tempo;
- Uma operação **update** é uma operação de leitura e de escrita combinadas;
- Finalmente, uma operação **capture** executa uma atualização, mas salva o valor calculado nessa região para ser utilizada no

código seguinte à região.

Se nenhuma cláusula for definida, uma operação **update** é assumida. A sua sintaxe é apresentada a seguir.

#pragma acc atomic [cláusula-atomic]

Um histograma é uma técnica comum para contar quantas vezes os valores ocorrem em um conjunto de entrada de acordo com o seu valor. No código listado a seguir, uma série de números inteiros de um intervalo conhecido é percorrida e conta-se o total de ocorrências de cada número nesse intervalo. Como cada número no intervalo pode ocorrer várias vezes, precisamos assegurar que cada elemento no vetor de histograma seja atualizado atomicamente. O código do exemplo em questão demonstra o uso da diretiva **atomic** para gerar um histograma.

```
#pragma accdata copyin(a[0:N]) copyout(h[0:HN])
for(int it = 0; it < ITERS; it++) {

#pragma acc parallel loop
    for (int i = 0; i < HN; i++)
         h[i] = 0;
#pragma acc parallel loop
    for (int i = 0; i < N; i++) {
#pragma acc atomic update
         h[a[i]] += 1; }
    }
```

Observe que as atualizações no vetor do histograma Math input error são executadas atomicamente. Como estamos incrementando o valor do elemento de um vetor, uma operação **update** é usada para ler o valor, modificá-lo e gravá-lo novamente.

O exemplo de cálculo dos graus dos vértices de um grafo é revisitado aqui com a versão em OpenACC e uso da diretiva **atomic**, conforme mostrado na listagem a seguir. Note que sem o uso da diretiva **atomic** o programa não produz resultados corretos, por

conta da condição de corrida ao incrementar os elementos do vetor *grau*.

```
#include <stdio.h>
#define ARESTAS 10
#define VERTICES 7
int main(int argc, char *argv[]) { /* acc_atomic.c  */
int i,j;
int num_arestas = ARESTAS, grau[VERTICES]={0,0,0,0,0,0,0};
typedef struct {
   int vertice1, vertice2;
   } tipo_aresta;
tipo_aresta aresta[ARESTAS];
    /* Atribui os valores iniciais aos vértices das arestas do
grafo */
    aresta[0].vertice1 = 0;
    aresta[0].vertice2 = 1;
    aresta[1].vertice1 = 0;
    aresta[1].vertice2 = 2;
    aresta[2].vertice1 = 1;
    aresta[2].vertice2 = 2;
    aresta[3].vertice1 = 1;
    aresta[3].vertice2 = 3;
    aresta[4].vertice1 = 2;
    aresta[4].vertice2 = 3;
    aresta[5].vertice1 = 2;
    aresta[5].vertice2 = 5;
    aresta[6].vertice1 = 2;
    aresta[6].vertice2 = 4;
    aresta[7].vertice1 = 3;
    aresta[7].vertice2 = 5;
    aresta[8].vertice1 = 3;
    aresta[8].vertice2 = 4;
    aresta[9].vertice1 = 4;
    aresta[9].vertice2 = 6;
    /* Executa em paralelo dividindo as iterações entre as
gangues */
    #pragma acc parallel loop gang
    for (j = 0; j< num_arestas; j++){
        #pragma acc atomic update
    /* A operação de incremento é realizada atomicamente */
        grau[aresta[j].vertice1]++;
```

```
            #pragma acc atomic update
        /* A operação de incremento é realizada atomicamente */
            grau[aresta[j].vertice2]++;
        }
        /* Imprime o resultado */
        for (i = 0; i < VERTICES; i++)
            printf("Grau do vértice %d = %d \n",i, grau[i]);
        return(0);
}
```

7.7 Funções OpenACC

Para que as funções da interface de aplicação do OpenACC possam ser utilizadas, deve-se incluir a seguinte definição:

```
#include <openacc.h>
```

A compilação condicional para preservar a portabilidade do código é suportada usando a variável de pré-processador **_OPENACC**.

As funções disponíveis são as seguintes:

- **acc_get_num_devices(devicetype)** — Retorna o número de dispositivos aceleradores do tipo especificado.
- **acc_set_device_type(devicetype)** — Define o tipo de dispositivo acelerador para ser usado por esta *thread* no hospedeiro.
- **acc_get_device_type()** — Retorna o tipo de dispositivo acelerador que está sendo utilizado por esta *thread* no hospedeiro.
- **acc_set_device_num(devicenum, devicetype)** — Define o número do dispositivo acelerador para ser usado por esta *thread* no hospedeiro.

- **acc_get_device_num(devicetype)** — Retorna o número do dispositivo acelerador para ser usado por esta *thread* no hospedeiro.

- **acc_async_test(expression)** — Retorna um valor não zero ou .TRUE. se todas as atividades assíncronas com a dada expressão foram completadas; senão retorna zero ou .FALSE.

- **acc_async_test_all()** — Retorna um valor não zero ou .TRUE. se todas as atividades assíncronas terminaram; senão retorna zero ou .FALSE.

- **acc_async_wait(expression)** — Espera até todas as atividades assíncronas com a expressão dada tenham terminado.

- **acc_async_wait_all()** — Espera até que todas as atividades assíncronas tenham terminado.

- **acc_init(devicetype)** — Inicia o sistema de execução e define o tipo de acelerador para ser usado por esta *thread* no hospedeiro.

- **acc_shutdown(devicetype)** — Desconecta esta *thread* hospedeira do dispositivo acelerador.

- **acc_on_device(devicetype)** — Em uma região **parallel ou kernels** do OpenACC, isso é utilizado para tomar um caminho de execução diferente dependendo se o programa está executando no acelerador ou no hospedeiro.

- **acc_malloc(size_t)** — Retorna o endereço de memória alocado no dispositivo acelerador.

- **acc_free(void*)** — Libera a memória alocada por **acc_malloc**.

O valor para o parâmetro *devicetype* é dependente de implementação. Alguns valores possíveis são *acc_device_default, acc_device_host e acc_device_nvidia*. Por favor consulte a seção 8.3.3 para uma lista completa das opções, de acordo com o compilador utilizado.

Uma possível utilização dessas funções é o uso de aceleradores em conjunto com o modelo de programação em memória distribuída com MPI. Nesse caso cada nó precisa descobrir quantos aceleradores possui, e associá-los ao processo em execução. Vejamos seu uso no exemplo a seguir.

```
#ifdef _OPENACC
#include <openacc.h>  /* Biblioteca OpenACC */
#endif
#include <mpi.h>
int main(int argc, char *argv[]) { /* acc_funcoes.c */
int meu_ranque;
    MPI_Init(&argc, &argv);
    /* Ranque MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
#ifdef _OPENACC/* Chamada OpenACC */
    /* Chamada OpenACC */
    acc_init(acc_device_nvidia);
    /* Determina o número de GPUS  */
    const int num_dev = acc_get_num_devices(acc_device_nvidia);
    /* Atribui uma GPU para cada processo MPI */
    const int dev_id = meu_ranque % num_dev;
    acc_set_device_num(dev_id, acc_device_nvidia);
      printf ("O processo MPI %d é atribuído à GPU %d \n",
meu_ranque, dev_id);
#endif
    MPI_Finalize();
}
```

E claro, também pode ser utilizado com o modelo de programação do OpenMP, como no próximo exemplo.

```
#ifdef _OPENACC
#include <openacc.h> /* Biblioteca OpenACC */
```

```
#endif
#include <omp.h>
int main(int argc, char *argv[]) { /* acc_funcoes2.c */
#pragma omp parallel num_threads(2)
{
    int tid = omp_get_thread_num();
#ifdef _OPENACC
    /* Chamada OpenACC */
    acc_init(acc_device_nvidia);
    /* Determina o número de GPUs */
    const int num_dev = acc_get_num_devices(acc_device_nvidia);
    const int dev_id = tid%num_dev;
    /* Atribui uma GPU para cada thread OpenMP */
    acc_set_device_num(dev_id, acc_device_nvidia);
      printf ("A thread %d é atribuída à GPU %d \n", tid,
dev_id);
#endif
}
}
```

7.8 Variáveis de ambiente

Nesta seção apresentamos algumas variáveis de ambiente que modificam o comportamento das regiões do acelerador. Os nomes devem ser todos em maiúsculas. Os valores atribuídos às variáveis de ambiente são insensíveis à caixa das letras e podem ter espaços em branco no seu início ou final. Se as variáveis de ambiente tiverem os seus valores modificados depois do início do programa, ou mesmo se o programa modificar esses valores, o comportamento é dependente de implementação.

ACC_DEVICE_TYPE

A variável de ambiente **ACC_DEVICE_TYPE** controla o dispositivo padrão a ser usado quando as regiões paralelas ou *kernels* forem executadas no acelerador, se o programa foi compilado com mais de um tipo diferente de dispositivo. Os valores permitidos são

dependentes de implementação, sendo que a documentação do compilador deve ser consultada para determiná-los. A chamada à função **acc_set_device_type** se sobrepõe ao uso dessa variável de ambiente. Por exemplo:

```
$ export ACC_DEVICE_TYPE=NVIDIA
```

ACC_DEVICE_NUM

A variável de ambiente **ACC_DEVICE_NUM** controla o número do dispositivo padrão quando executando regiões no acelerador. O valor dessa variável de ambiente deve ser um inteiro não negativo entre 0 e o número de dispositivos presentes no hospedeiro. Se o valor for maior ou igual ao número de dispositivos existentes, o comportamento é dependente de implementação. Uma chamada à função **acc_set_device_num** se sobrepõe ao valor definido nessa variável. Por exemplo:

```
$ export ACC_DEVICE_NUM=1
```

ACC_PROF_LIB

A variável de ambiente **ACC_PROF_LIB** especifica a biblioteca de *profile* a ser utilizada. Por exemplo:

```
$ export ACC_PROFLIB=/path/to/proflib/libaccprof.so
```

7.9 Erros comuns e recomendações

A seguir listamos alguns erros e omissões comuns a programadores menos experientes e elencamos algumas recomendações para serem utilizadas na programação com o OpenACC.

Algumas omissões frequentes

- Não se deve esquecer de utilizar a cláusula **atomic** para acessar variáveis que são compartilhadas por mais de uma *thread* (gang, worker ou vector) simultaneamente.

- O compilador deve ser capaz de identificar o tamanho das regiões de dados, o que é possível com o uso adequado dos parâmetros da diretiva **data**. Deve-se especificar a dimensão dos vetores sendo movimentados, por exemplo, **copyin (a[0:n], b[0:n])**.

- O uso de chamadas de função em regiões paralelas no acelerador requer o uso da diretiva **routine** quando da declaração ou no protótipo da rotina. E deve-se fazer uso de regiões de dados (diretiva **data**) para evitar a movimentação desnecessária de dados entre o hospedeiro e o acelerador.

- Um erro comum é a omissão da diretiva **parallel**, o que vai resultar na execução sequencial do trecho de código:

```
{ // sem diretiva "parallel"
#pragma acc
    for (size_t i = 0; i < n; ++i ) {
...
    }
}
```

- Uma troca frequente é o uso palavra *for* no lugar da palavra *loop*, o que vai ocasionar um erro na compilação do código.

```
{ // uso do "for" no lugar de "loop"
#pragma acc parallel for
    for (size_t i = 0; i < n; ++i ) {
...
    }
}
```

Algumas recomendações

- Os laços aninhados são melhores para a paralelização pelo OpenACC, sendo que as iterações dos laços devem ser independentes entre si. Para auxiliar o compilador, recomenda-se o uso da palavra-chave *restrict* para os ponteiros e da cláusula **independent** nos laços em uma região *kernels*.

- Cuidado para utilizar corretamente a palavra-chave, não confundir *float restrict *ptr* (incorreto) com *float *restrict ptr* (correto).

- Com o compilador PGI, use a opção *time* para que seja impresso o tempo gasto em cada rotina e na transferência de dados:

  ```
  -ta=nvidia,time
  ```

- O uso de aritmética com ponteiros deve ser evitado, com a transformação das referências aos dados para arranjos indexados, sendo que regiões contíguas de memória devem ser usadas para os arranjos multidimensionais.

- Não há necessidade de se utilizar a cláusula **private** ou **fisrtprivate** para variáveis escalares acessada dentro de um laço, já que elas são definidas como *firstprivate* por padrão.

Outros erros comuns

- Existem cláusulas que têm a mesma sintaxe tanto no OpenMP quando no OpenACC. Uma delas é a **reduction**. Os parâmetros de uso e observações são os mesmos em ambos os paradigmas.

- Um erro comum é a omissão do tipo de dispositivo que será utilizado na cláusula **device_type**.

- Especificar as cláusulas **num_gangs**, **num_workers** e **vector_length** com valores inadequados, ou muito pequenos,

prejudicando o paralelismo; ou que excedem a capacidade de *threads* suportada pelo acelerador utilizado.

- Especificar uma variável na cláusula **reduction** com um operador diferente da operação aritmética utilizada dentro no laço paralelizado, por exemplo, utilizando soma quando a operação é de multiplicação, ou vice-versa.

- Especificar as cláusulas **gang**, **worker** ou **vector** com parâmetros numéricos dentro de uma região *parallel* ou dentro de uma região *kernels* com as cláusulas **num_gangs**, **num_workers** e **vector_length** já especificadas.

7.10 Estudos de caso

Após a introdução do conceito e das principais diretivas usadas no OpenACC, veremos alguns exemplos de aplicações dessas diretivas, como elas se comportam e quais as melhores opções de uso para obter-se um melhor desempenho.

Os códigos foram executados em um servidor com dois processadores Intel Xeon E5-2609 (2,40 GHz, 4 núcleos cada, cache de 10 MB), com 128 GB de memória compartilhada, discos locais de alta velocidade SSD (Solid-State Drive) e um acelerador NVIDIA Tesla K80 (24 GB de memória, 4992 CUDA cores). O sistema operacional usado foi a versão 7.3 da distribuição Centos Linux de 64 bits. Todos os códigos foram compilados usando o compilador PGI Community Edition Version 20.1.

7.10.1 Estudo de caso: cálculo de Pi

O primeiro estudo de caso a ser realizado é de um problema intrigante há séculos na história da matemática: o cálculo do número Pi. O valor de Pi é definido pela relação entre o perímetro de uma circunferência e seu diâmetro e, para a maioria dos cálculos simples,

é comum usar a aproximação do valor de Pi para 3,1415, obtida usualmente por aproximações utilizando polígonos inscritos e circunscritos ao círculo.

Contudo, para cálculos científicos mais rigorosos, é necessário que esse valor seja calculado com uma precisão muito maior. Em computação existe uma grande variedade de algoritmos para o cálculo do valor aproximado de Pi, tais como o algoritmo Gauss-Legendre[2] e o método de Monte Carlo[3]. No exemplo a seguir é apresentada uma implementação simples para o cálculo sequencial (serial) de Pi a partir da integração numérica usando o método do ponto médio (ou dos retângulos) da seguinte fórmula:

Math input error

Essa equação é derivada a partir da seguinte igualdade trigonométrica:

Math input error

Nessa versão sequencial, a execução do cálculo de Pi dentro do laço é feita por uma única *thread* ou processo, assim sendo, o programa será executado mais rapidamente ou não exclusivamente em função da velocidade de execução de um único processador.

```
#include <stdio.h>
#define N 10000000000
int main(int argc, char *argv[]) { /* calcpi_seq.c  */
double pi = 0.0f;
long i;
    for (i = 0; i < N; i++) {
         double t = (double) ((i+0.5)/N);
         pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n",pi/N);
    return(0);
}
```

Para que o cálculo seja executado pelo acelerador, usaremos a diretiva **parallel** do OpenACC e adicionaremos apenas uma linha antes do laço:

#pragma acc parallel loop reduction(+: pi).

As iterações do laço são divididas entre as gangues e enviadas para execução no acelerador. Ao final, os valores parciais calculados pelas diversas gangues são totalizados com uma operação de redução no acelerador e o resultado é enviado de volta para o processador principal.

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[]) { /* acc_calcpi.c  */
double pi = 0.0f;
long N=10000000000;
    /* Usa rotina do OpenMP para medir o tempo */
    double inicio = omp_get_wtime();
    /* Executa as iterações em paralelo
    e faz a redução ao final */
    #pragma acc parallel loop reduction(+: pi)
    for (long i = 0; i < N; i++) {
         double t=(double) ((i+0.5)/N);
         pi += 4.0/(1.0+t*t);
    }
    double fim = omp_get_wtime();
    /* Imprime o resultado */
    printf("O valor de pi é: %f\n",pi/N);
    printf("O tempo de execução é %f segundos.\n",fim-inicio);
    return(0);
}
```

Para avaliar os ganhos no tempo de execução do cálculo de Pi em cada caso, foram executadas duas versões de código: sequencial e com OpenACC. O tempo de execução sequencial do cálculo de Pi foi de 47,60 segundos, o tempo com o uso do OpenACC o tempo de execução foi de 0,60 segundos.

O *speedup* encontrado usando programação OpenACC foi de 79,30 em comparação com a execução em sequencial. Os resultados são apresentados na Figura 7.6.

Tabela 7.2: Cálculo de Pi com N = Math input error

Tempo Sequencial	**Tempo Paralelo**	**Speedup**
Math input error s	Math input error s	Math input error

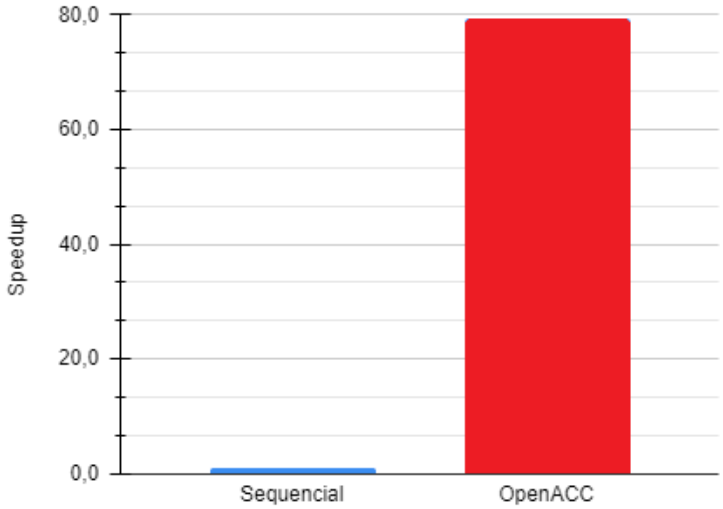

Figura 7.6: Speedup - Cálculo do valor de Pi

0 *speedup* que foi obtido é resultado de dois fatores principais:

1. O grande número de núcleos existentes no acelerador utilizado, perto de 5000 *cuda cores*. Embora a capacidade de processamento, individualmente, seja bem menor que um processador convencional, em conjunto eles oferecem um desempenho que neste caso é dezenas de vezes superior.

2. A movimentação de dados, neste caso, não tem grande influência nos tempos de execução no acelerador, pois trata-se apenas de um valor escalar, cujo tempo de transferência entre o hospedeiro e o acelerador é irrelevante.

A avaliação dos resultados com um acelerador é bastante distinta dos casos analisados até agora com o uso de processadores

convencionais. Assim, outras métricas devem ser consideradas e outros fatores avaliados para verificarmos se um problema foi adequadamente paralelizado.

Outros parâmetros que merecem alguma atenção são a taxa de ocupação dos núcleos e da memória do acelerador, o que pode nos fornecer uma ideia do limite do ganho que pode ser obtido com algum tipo de problema. É comum que essa ocupação não chegue a 100% em função das limitações no paralelismo disponível no código e do *hardware* da GPU.

Para medir o uso de recursos do acelerador NVIDIA Tesla K80, variou-se o valor de Math input error no código. Os valores de Math input error usados para as medições foram entre Math input error a Math input error. Na Tabela 7.3, são apresentados os resultados obtidos para o tempo de processamento, uso de memória e percentual de uso de *GPU*.

Tabela 7.3: Uso de recursos do acelerador - Cálculo de Pi

Tamanho de N	**Tempo (s)**	**Memória (MB)**	**GPU (%)**
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error

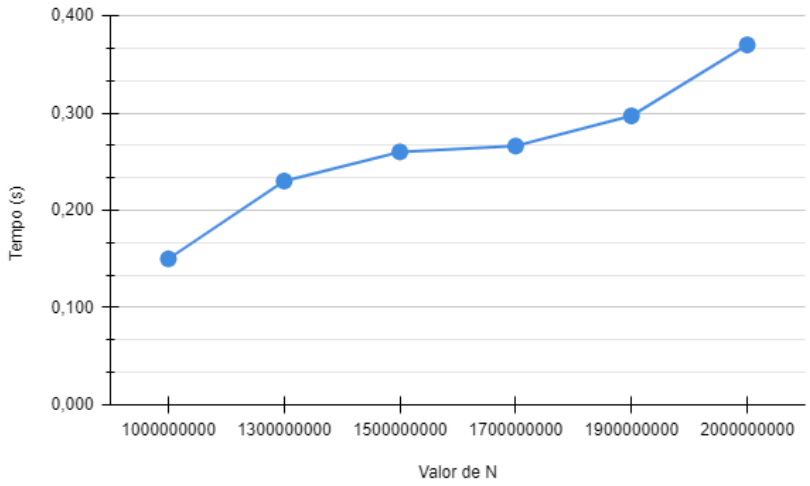

Figura 7.7: Percentual de uso da *GPU*

Verificamos que com o maior valor N utilizado, o percentual de uso de GPU não passa de 50%, o que indica que o acelerador teria capacidade para resolver problemas ainda maiores, o que provavelmente resultaria em valores de *speedup* também maiores em relação à execução sequencial (Figura 7.7).

Outro indicativo para essa afirmação é que o uso de memória do acelerador também ficou abaixo da capacidade máxima de memória. Isso é de certa forma esperado pois o código é bem simples, possuindo apenas variáveis escalares (Figura 7.8).

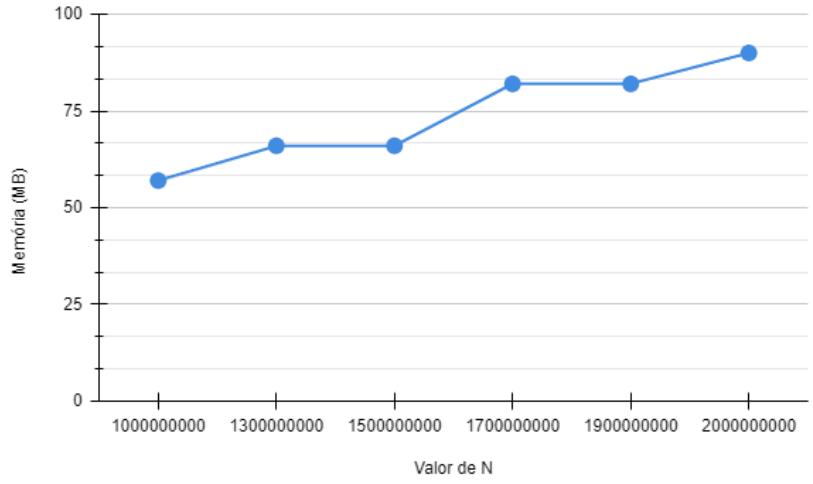

Figura 7.8: Uso de memória

7.10.2 Estudo de caso: método do trapézio

Para a execução do código em OpenACC, adicionaremos a linha **#pragma acc parallel loop reduction(+:integral)** antes do laço, transferindo a execução do laço para o acelerador.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

double f(double x) {
double valor;
    valor = exp(x);
    return(valor);
}

int main(int argc, char *argv[]) {/* acc_trapezio.c */
double integral;    /* Armazena resultado */
double a, b;        /* Limite esquerdo e direito */
long i,n;           /* Número de trapezóides */
double h;           /* Largura da base do trapezóide */
    a = 0.0;
    b = 1.0;
    n = 800000000;
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
     /* Vamos usar a função do OpenMP */
    double inicio = omp_get_wtime();
#pragma acc parallel loop reduction(+:integral)
    for (i = 1; i < n-1; i++) {
        integral += f(a + i*h);
    }
    integral *= h ;
    printf("Com n = %ld trapezoides, a estimativa \n", n);
       printf("da integral de %f ate %f = %lf \n", a, b,
integral);
    printf("Tempo: \t %f \n", omp_get_wtime()-inicio);
    return(0);
}
```

Para se obter um melhor desempenho do código em OpenACC, a operação de cálculo da integral será movimentada para o acelerador, deixando de ser executada no hospedeiro. Para isso será usada a diretiva **data** do OpenACC. Adicionar a linha **#pragma acc data copy(integral)** antes linha de execução da integral para fazer a cópia da variável *integral* para a memória do acelerador.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

#pragma acc routine
double f(double x) {
double valor;
    valor = exp(x);
    return(valor);
}

int main(int argc, char *argv[]) { /* acc_trapezio2.c  */
double integral;    /* Armazena resultado */
double a, b;        /* Limite esquerdo e direito */
long i,n;           /* Número de trapezóides */
double h;           /* Largura da base do trapezóide */

    a = 0.0;
    b = 1.0;
    n = 800000000;
    h = (b-a)/n;
#pragma acc data copy(integral)
    integral = (f(a) + f(b))/2.0;
    /* Vamos usar a função do OpenMP */
    double inicio = omp_get_wtime();
#pragma acc parallel loop reduction(+:integral)
    for (i = 1; i < n-1; i++) {
        integral += f(a + i*h);
    }
    integral *= h ;
    printf("Com n = %ld trapezoides, a estimativa \n", n);
       printf("da integral de %f ate %f = %lf \n", a, b,
integral);
    printf("Tempo: \t %f \n", omp_get_wtime()-inicio);
```

```
    return(0);
}
```

Com a nova versão do código movimentando a execução da integral para o acelerador o *speedup* aumentou para 119 em comparação ao código sequencial. Na primeira versão do código em OpenACC, o *speedup* foi de 18 em comparação ao código sequencial. Os resultados são apresentados na Figura 7.9.

Assim como no cálculo de Pi, no método do trapézio também se variou o valor de N para fazer as medições de uso dos recursos do acelerador. Os valores de Math input error usados para as medições variaram entre Math input error a Math input error. Os resultados obtidos para o tempo de processamento, uso de memória e percentual de uso de GPU são apresentados na Tabela 7.4.

Tabela 7.4: Uso de recursos do acelerador- Trapezio

Tamanho de N	**Tempo (s)**	**Memória (MB)**	**GPU (%)**
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error

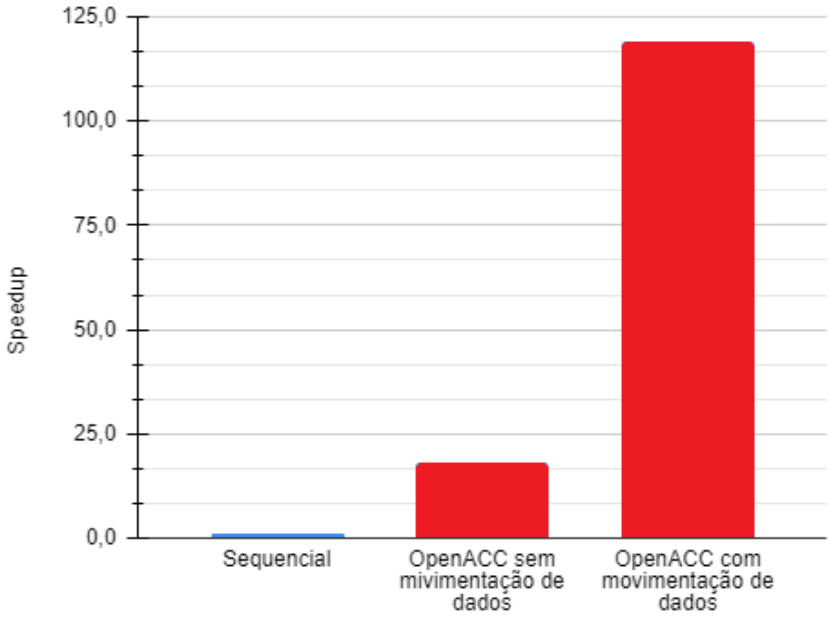

Figura 7.9: Speedup - Método do Trapézio

Em relação ao uso de *GPU*, houve um aumento linear entre os valores de Math input error de Math input error a Math input error de utilização da *GPU*. Durante o processo de execução dos valores de Math input error e Math input error a *GPU* teve 100% de utilização, conforme a Figura 7.10.

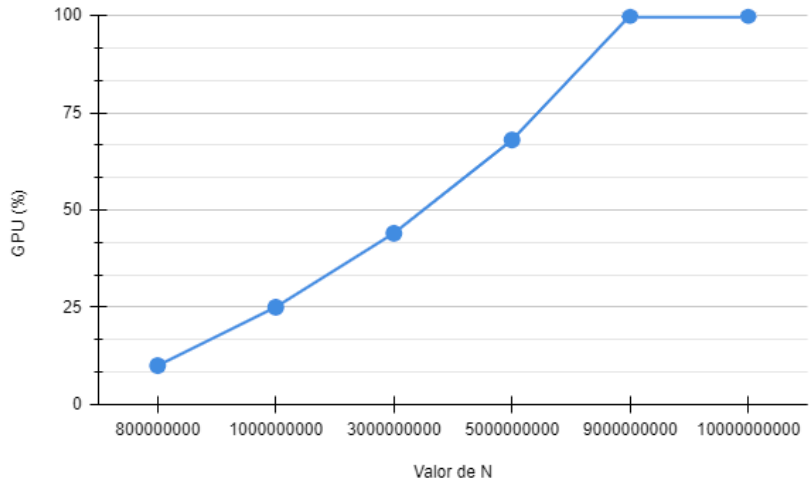

Figura 7.10: Percentual de uso da *GPU*

7.10.3 Estudo de caso: método Jacobi

O Método de Jacobi é um procedimento iterativo para a resolução de sistemas lineares. Neste exemplo faremos o cálculo da temperatura na placa usando a equação de Laplace: Math input error. Essa equação converge iterativamente para o valor correto, calculando novos valores em cada ponto a partir da média dos pontos vizinhos, como visto na equação a seguir.

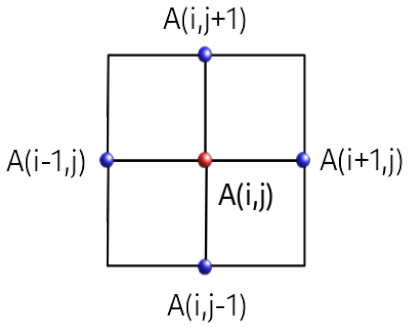

Figura 7.11: Malha do método Jacobi

Math input error

A implementação sequencial (serial) para o cálculo da temperatura da placa usando o método Jacobi é apresentada a seguir.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define  COLUMNS        1000
#define  ROWS           1000
#define  MAX_TEMP_ERROR 0.01
double Anew[ROWS+2][COLUMNS+2];
double A[ROWS+2][COLUMNS+2];
void initialize();
void track_progress(int iter);
int main(int argc, char *argv[]) {
    int i, j;
    int max_iterations=1000;
    int iteration=1;
    double dt=100;
    initialize();
    while ( dt > MAX_TEMP_ERROR && iteration
```

```
            <= max_iterations ) {
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Anew[i][j] = 0.25 * (A[i+1][j] +
                A[i-1][j] + A[i][j+1] + A[i][j-1]);
            }
        }
        dt = 0.0;
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
            }
        }
        iteration++;
    }
    printf("\n Erro maximo na iteracao %d era %f\n",
           iteration-1, dt);
}
void initialize(){
    int i,j;
    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            A[i][j] = 0.0;
        }
    }
    for(i = 0; i <= ROWS+1; i++) {
        A[i][0] = 0.0;
        A[i][COLUMNS+1] = (100.0/ROWS)*i;
    }
    for(j = 0; j <= COLUMNS+1; j++) {
        A[0][j] = 0.0;
        A[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

Como pode ser visto no código sequencial, o primeiro laço dentro do *while* de convergência calcula o novo valor para cada elemento com base nos valores atuais de seus vizinhos, cujo resultado é armazenado em uma matriz temporária **Anew**. Isso garante que todos os valores sejam calculados usando o estado atual de **A** antes

que o conteúdo de **A** seja novamente atualizado. Como resultado, cada iteração do laço é completamente independente da outra.

Esse laço também calcula um máximo valor de erro. O valor do erro é a máxima diferença de temperatura entre o novo valor e o antigo. Se o erro entre duas iterações estiver dentro de alguma tolerância, o problema será considerado como convergido e o laço externo será encerrado. O segundo laço simplesmente atualiza o valor de **A** com os valores calculados em **Anew**.

Para que o cálculo seja executado pelo acelerador usaremos a diretiva **parallel** do OpenACC. Adicionar a linha **#pragma acc parallel loop**, e a linha **#pragma acc parallel loop reduction(max:dt)** no segundo laço.

```
#pragma acc parallel loop   /* acc_jacobi1.c   */
    for (i = 1; i <= ROWS; i++)
         for (j = 1; j <= COLUMNS; j++) {
              Anew[i][j] = 0.25 * (A[i+1][j] +
              A[i-1][j] + A[i][j+1] + A[i][j-1]);
        }
    dt = 0.0;
#pragma acc parallel loop reduction(max:dt)
    for (i = 1; i <= ROWS; i++)
         for (j = 1; j <= COLUMNS; j++) {
              dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
              A[i][j] = Anew[i][j];
        }
```

Para avaliar os tempos de execução dos cálculos da temperatura, foram executadas as versões sequencial e com OpenACC.

Foram observados os seguintes tempos de execução: o código sequencial teve um tempo total de 11 segundos, o tempo de execução com o OpenACC foi de 10 segundos. O *speedup* encontrado na execução do código com OpenACC foi de 1,10 em comparação com a execução em sequencial. Os resultados são apresentados na Figura 7.12.

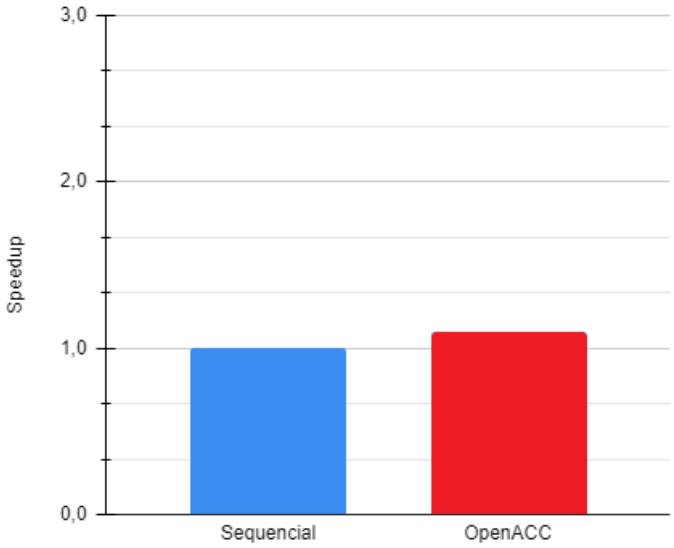

Figura 7.12: Speedup - Método Jacobi

Como visto, o tempo de execução com uso OpenACC foi quase idêntico ao tempo de execução do código sequencial. O principal motivo para isso é o fato da matriz de cálculo não estar armazenada no acelerador. Assim, toda vez que o acelerador executa uma operação, os resultados são gravados na memória do hospedeiro, o que tem um alto custo computacional, tornando a execução do código mais lenta.

Para resolver esse problema é necessário fazer a cópia da matriz **A** para o acelerador no início da região paralela. Adicionalmente, a matriz **Anew** será criada exclusivamente no acelerador. Para isso, adicionaremos a linha **#pragma acc data copy(A) create (Anew)** antes dos dois laços.

```
#pragma acc data copy(A) create(Anew)   /* acc_jacobi2.c   */
     while (dt > MAX_TEMP_ERROR && iteration <= max_iterations)
{
#pragma acc parallel loop
        for (i = 1; i <= ROWS; i++)
             for (j = 1; j <= COLUMNS; j++) {
                  Anew[i][j] = 0.25 * (A[i+1][j] +
                  A[i-1][j] + A[i][j+1] + A[i][j-1]);
             }
        dt = 0.0;
```

```
#pragma acc parallel loop reduction(max:dt)
        for (i = 1; i <= ROWS; i++)
             for (j = 1; j <= COLUMNS; j++) {
                  dt = fmax( fabs(Anew[i][j]-A[i][j]), dt);
                  A[i][j] = Anew[i][j];
             }
```

O total de tempo de execução sequencial foi de 11 segundos e o tempo de execução com OpenACC utilizando movimentação de dados foi reduzido para 0,80 segundos. Com a nova versão do código o *speedup* aumentou para 13,80 em relação à versão sequencial.

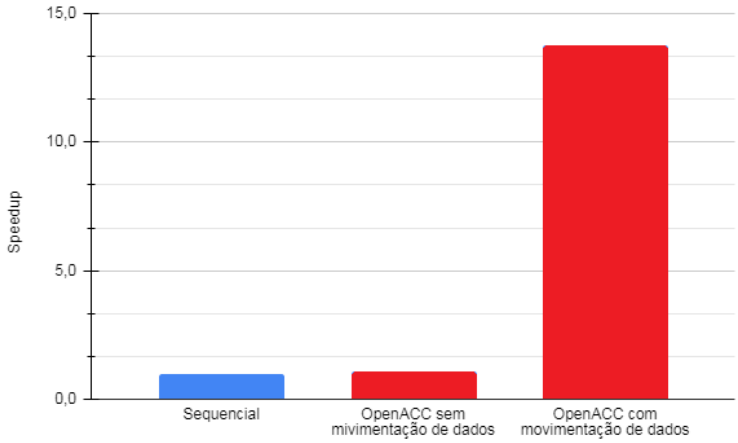

Figura 7.13: Speedup - Método Jacobi

Esse é um exemplo onde o uso de movimentação de dados propicia ganhos consideráveis. Os resultados são apresentados na Figura 7.13. Para medir o uso de recursos do acelerador NVIDIA Tesla K80, variou-se o tamanho da matriz no código. Os tamanhos das matrizes usadas para as medições foram entre Math input error a Math input error. Na Tabela 7.5, são apresentados os resultados obtidos para o tempo de processamento, uso de memória e percentual de uso de *GPU*.

Tabela 7.5: Uso de recursos do acelerador - Jacobi

Tamanho da Matriz	**Tempo (s)**	**Memória (MB)**	**GPU (%)**

Tamanho da Matriz	Tempo (s)	Memória (MB)	GPU (%)
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error
Math input error	Math input error	Math input error	Math input error

Conforme a Figura 7.14 e a Figura 7.15, os valores obtidos durante o tempo de execução e uso de memória tiveram um comportamento de utilização dos recursos próximos ao linear. Por sua vez, o uso de *GPU* sempre se manteve próximos ao percentual de 100% de uso. Isso ocorre porque toda a matriz é processada no acelerador (Figura 7.16).

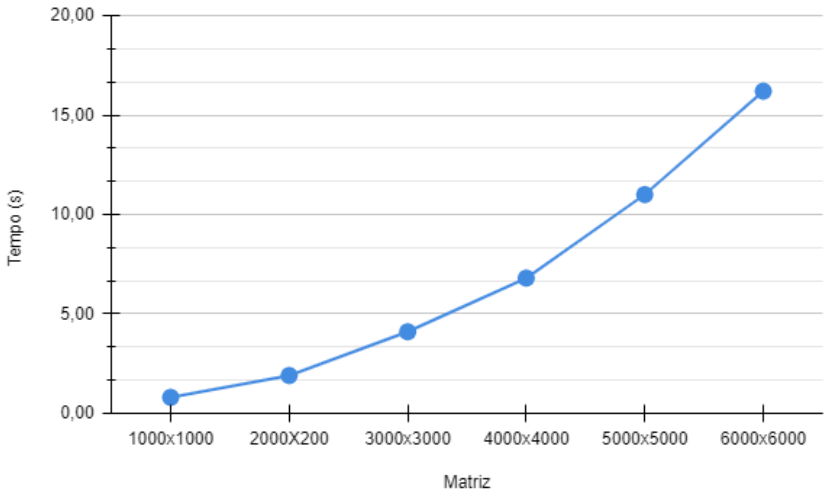

Figura 7.14: Tempo de execução

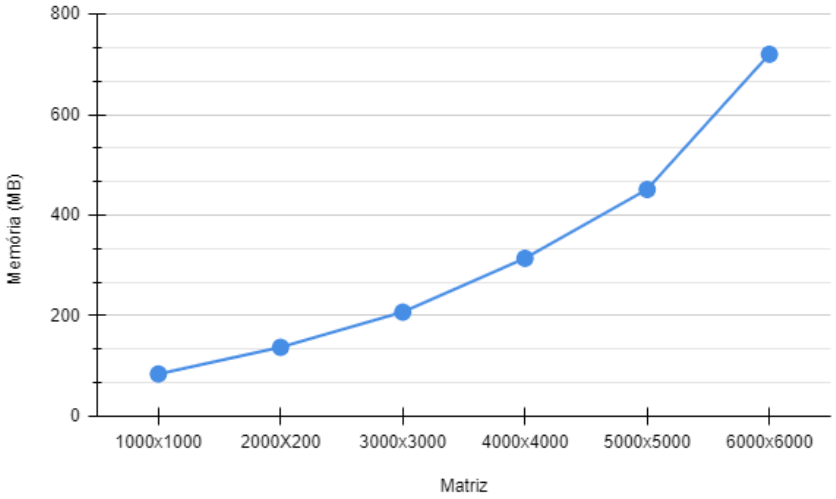

Figura 7.15: Uso de memória

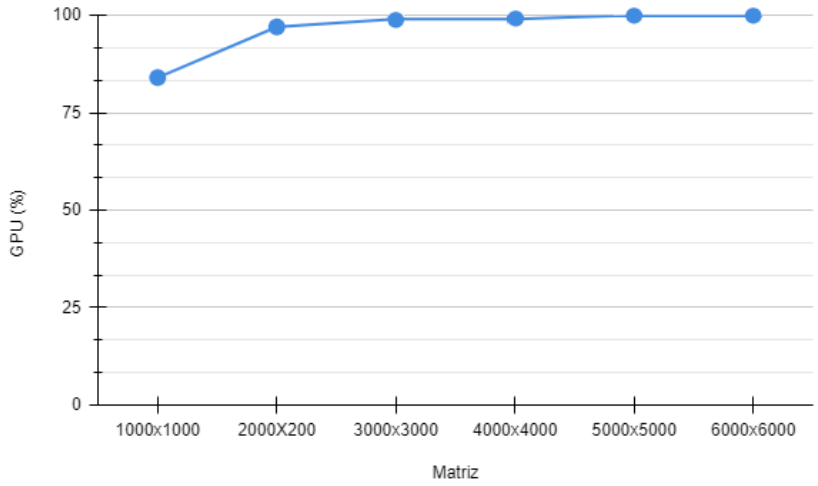

Figura 7.16: Percentual de uso da *GPU*

7.11 Exercícios

1. Com relação ao conceito de OpenACC, responda:
    1. Qual o principal objetivo do uso do OpenACC.
    2. Quais são as principais características do OpenACC.
    3. Descreva o que são diretivas.

4. Descreva o que são cláusulas.

2. Por que o gerenciamento das transferências de dados entre as memórias da GPU e da CPU é importante?

3. Quais são os passos do ciclo de desenvolvimento e análise de código na programação paralela?

4. Descreva os três níveis de paralelismo do OpenACC?

5. Dado um kernel que é executado com 4 gangues, cada uma com 2 trabalhadores com 256 vetores de comprimento, qual o número total de *threads*?

6. Como se comparam os níveis de paralelismo do OpenACC com os do CUDA?

7. Qual o objetivo do uso dos comandos nvidia-smi e pgaccelinfo?

8. Descreva as características das diretivas **kernels e parallel**. Quais as condições para usá-las e quais as suas diferenças?

9. Quais são os comandos de movimentação de dados, e quando devemos utilizá-los?

10. Com relação à movimentação de dados, descreva quais são as suas principais cláusulas e suas condições de uso.

11. Reescreva o laço seguinte, utilizando as cláusulas de movimentação de dados.

```
#pragma acc parallel loop
{
   for (int i = 0; i < n; i++) {
        c[i] = a[i] * b[i]
      }
}
```

12. Descreva as caraterísticas das cláusulas a seguir e um exemplo de seu uso.

    1. device_type
    2. vector_length
    3. tile
    4. collapse
    5. atomic

13. Considere o código mostrado no exemplo de "Movimentação de dados" da Seção 7.3.1. Compile e execute esse código, verificando o tempo total de execução. Pergunta-se: como melhorar o desempenho desse código através do aumento do paralelismo e da diminuição da movimentação de código? Qual o novo tempo de execução obtido? Qual o speedup?

14. Execute os seguintes códigos e responda se os resultados são os mesmos. Por quê?

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
float A[1000];

   #pragma acc kernels
   for(int iter = 1; iter < 1000; iter++){
      A[iter] = 1.0;
   }
   A[10] = 2.0;
   printf("A[10] = %f\n", A[10]);
   return(0);
}
```

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
float A[1000];

#pragma acc data copy(A){
```

```
    #pragma acc kernels
    for(int iter = 1; iter < 1000; iter++){
       A[iter] = 1.0;
    }
    A[10] = 2.0;
}
    printf("A[10] = %f\n", A[10]);
    return(0);
}
```

15. Modifique o código apresentado a seguir:

    1. Use as diretivas de kernels e parallel e verifique qual obteve o melhor desempenho;

    2. Use diretivas de movimentação de dados;

    3. Altere o tamanho do vetor e bloco e verifique o desempenho final;

    4. Use a diretiva **device_type**.

```
#include <stdio.h>
#define SIZE 5000

float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

int main(int argc, char *argv[]) {
int i,j,k;
    for (i = 0; i < SIZE; ++i) {
         for (j = 0; j < SIZE; ++j) {
              a[i][j] = (float)i + j;
              b[i][j] = (float)i - j;
              c[i][j] = 0.0f;
         }
    }
    for (i = 0; i < SIZE; ++i) {
         for (j = 0; j < SIZE; ++j) {
              c[i][j] = a[i][j] + b[i][j];
```

8 Ambientes de execução

8.1 Preparação do ambiente de execução MPI

Apresentamos aqui um breve roteiro sobre a instalação e configuração da biblioteca MPI, assim como executar um programa MPI em algumas distribuições de sistemas operacionais Posix.

8.1.1 Introdução

Primeiro devemos considerar que há várias implementações disponíveis para o MPI, algumas gratuitas e outras comerciais. O processador e a rede de interconexão utilizados pelo sistema onde as aplicações serão executadas são alguns dos fatores a serem considerados na escolha da distribuição adequada de MPI. Eventualmente, algumas distribuições suportam mais de um tipo de processador ou de interface de rede, o que lhes confere uma possibilidade maior de adequação às características do seu ambiente.

É importante observar que a arquitetura dos processadores muda de geração para geração, com a inclusão de novas instruções e facilidades. Uma tendência muito comum tem sido a melhoria da capacidade de processamento vetorial do processadores, o que tem grande impacto no desempenho final das aplicações científicas.

Alguns compiladores comerciais, como o Intel e PGI, podem gerar código otimizado para diferentes arquiteturas de processadores em um único executável, aliviando a necessidade de geração de códigos binários separados para cada tipo de processador. O compilador de

código aberto GNU só tem suporte para essa facilidade nas versões mais recentes. A seguir algumas das distribuições mais importantes:

- OpenMPI - Mantida por um conjunto de diversos parceiros da área acadêmica, de pesquisa e da indústria, incluindo AMD, Bull, Cisco Systems, Centro de Computação de Alto Desempenho de Stuttgart (HLRS), IBM, Intel, Laboratório Nacional de Los Alamos e NVIDIA. Disponível em https://www.open-mpi.org/

- MPICH - Mantida também por um conjunto de parceiros da área acadêmica, de pesquisa e da indústria. Tem como alvo sistemas de alto desempenho e suporta diversos compiladores, tais como GNU, LLVM Clang, Intel, PGI e Sun Studio. Disponível em http://www.mpich.org/

- MVAPICH2 - É uma implementação derivada da MPICH e mantida pela Universidade Estadual de Ohio voltada para aplicações de alto desempenho com uso de redes de interconexão tais como OpenFabrics-IB, Omni-Path, OpenFabrics-iWARP, PSM e TCP/IP. Disponível em http://mvapich.cse.ohio-state.edu/

- Intel® MPI Library - uma biblioteca de troca de mensagens que implementa a especificação MPICH de código aberto. Mantida e distribuída pela Intel. Possui uma versão gratuita para estudantes, acadêmicos e pesquisadores. Disponível em https://software.intel.com/en-us/qualify-for-free-software

O uso de programas paralelos adiciona um fator de complexidade relativo ao tipo de rede de interconexão utilizada pelo sistema de computação no qual o programa paralelo MPI vai ser executado. A maioria das implementações atuais do MPI inclui suporte para diversos tipos de rede em único programa executável, de modo que ele possa ser executado com diferentes tipos de rede de interconexão sem necessidade de recompilação.

Dependendo do seu equipamento, de sua aplicação e dos recursos disponíveis, uma implementação pode ser melhor do que a outra. Essas distribuições evoluem ao longo do tempo mas, no momento da redação deste livro, as características a seguir podem ser destacadas.

O OpenMPI é bastante flexível em termos de plataformas de *hardware*, com suporte para Infiniband, mas o desempenho global é ligeiramente inferior ao de outras implementações como Intel MPI e MVAPICH2. O OpenMPI também suporta a rede Cray Gemini, mas não é uma implementação oficialmente suportada pela Cray.

O MPICH é uma implementação de referência, sendo considerada uma das implementações mais populares do padrão MPI, servindo de base para uma série de implementações derivadas como IBM MPI, Cray MPI, Intel MPI e MVAPICH, entre outras. Contudo, o seu suporte para InfiniBand é precário, ao contrário do OpenMPI. Mas suas variantes como Intel MPI e o MVAPICH2 possuem um suporte bastante eficiente para InfiniBand. Se considerarmos o conjunto da MPICH e suas variantes, o suporte de rede é extremamente variado, incluindo tanto o InfiniBand e redes de interconexão proprietárias como Cray Seastar, Gemini e Áries, como o IBM Blue Gene (/L, /P e /Q).

O MVAPICH2 é otimizado para InfiniBand, mas ele não tem uma boa afinidade entre processos e cores em ambientes com múltiplas *threads*.

O Intel MPI tem bom desempenho e bastante flexibilidade de uso, quanto aos tipos de processadores e redes suportadas, mas é um produto comercial que requer a aquisição de uma licença de uso. Além da Microsoft, é a única implementação MPI com suporte para o sistema operacional Windows.

Em um eixo ortogonal ao suporte da plataforma de *hardware*, encontramos a cobertura do padrão MPI. Nesse sentido, a

implementação MPICH é de longe muito superior. O MPICH foi a primeira implementação a suportar cada lançamento do padrão MPI, desde a versão MPI-1 até a MPI-3. O OpenMPI só recentemente suporta a versão MPI-3 e ainda assim parcialmente (não tem suporte para o formato "external32" do MPI I/O, por exemplo) e apresenta problemas em algumas plataformas. Tanto o OpenMPI como o MPICH possuem suporte completo para o MPI_THREAD_MULTIPLE, ou seja, se o processo é "multithreaded", múltiplas *threads* podem usar a biblioteca MPI sem restrições.

Finalmente, em termos de gerenciamento de processos, onde o OpenMPI era bem melhor há algum tempo atrás, agora o novo gerenciador de processos do MPICH, o Hydra, é tão bom e tem a mesma usabilidade e facilidades que o ORTE, do OpenMPI. O velho gerenciado do MPICH, chamado MPD, era difícil de usar e sem muitas opções, mas foi tornado obsoleto já há alguns anos.

Maiores informações sobre o OpenMPI e MPICH podem ser obtidas em https://github.com/open-mpi/ompi/blob/master/README.md e https://www.mpich.org/documentation/guides/, respectivamente.

8.1.2 Instalação

Vamos considerar como instalar e configurar as duas distribuições mais importantes disponíveis como software de código aberto e gratuitas: OpenMPI e MPICH. Felizmente a maioria das distribuições Linux já conta pacotes disponíveis para instalação dessas versões, o que torna essa tarefa mais fácil.

Fedora, CentOS e similares

Basicamente, os diversos tipos de implementação MPI podem ser instalados diretamente a partir do repositório:

- **OpenMPI**

```
$ sudo dnf install openmpi
$ sudo dnf install openmpi-devel
```

- **MPICH**

```
$ sudo dnf install mpich
$ sudo dnf install mpich-devel
```

Ubuntu, Debian e similares

- **OpenMPI**

  Basicamente, instalar o OpenMPI do repositório:

```
$ sudo apt-get install openmpi-bin
$ sudo apt-get install openmpi-doc
$ sudo apt-get install libopenmpi-dev
```

- **MPICH**

  Instale o pacote MPICH diretamente do repositório:

```
$ sudo apt-get install mpich
$ sudo apt-get install mpich-doc
$ sudo apt-get install libmpich-devel
```

MacOS

- **OpenMPI**

  A instalação automatizada do OpenMPI é possível no MacOS com a instalação do programa *homebrew*, no endereço https://brew.sh/, com o seguinte comando:

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

  E então digitar o seguinte comando:

```
$ brew install open-mpi
```

- **MPICH**

  A instalação automatizada do MPICH é possível no MacOS com a instalação do programa *homebrew*, no endereço https://brew.sh/, com o seguinte comando:

  ```
  $ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
  ```

  E então digitar o seguinte comando:

  ```
  $ brew install mpich
  ```

8.1.3 Compilação

Para compilar um arquivo fonte prog.c, digite:

```
$ mpicc -o prog prog.c
```

Para compilar um arquivo em Fortran, os seguintes comandos são possíveis:

```
$ mpif77 -o prog prog.f
$ mpif90 -o prog prog.f90
```

8.1.4 Configurando o SSH

Para a execução em ambientes com um pequeno número de máquinas conectadas via rede, é importante configurar o ambiente para que não seja necessário o uso de senhas cada vez que executarmos um comando em outra máquina. Para isso, deve-se criar e configurar o uso de chaves SSH para todas as máquinas da rede. Neste exemplo, assumimos que existem 2 máquinas nomeadas como *maquina1* e *maquina2*, e um nome de usuário *gabriel*.

Primeiro, criamos o diretório $HOME/.ssh com o comando:

```
maquina1:~$ mkdir $HOME/.ssh
```

Em seguida, geramos uma chave púbica e uma chave privada com o comando:

```
maquina1:~$ ssh-keygen -t rsa -b 4096 -C "gabriel@exemplo.edu"
```

Pressione a tecla "Entre" para responder às perguntas seguintes com o padrão.

```
Enter file in which to save the key
(/home/gabriel/.ssh/id_rsa): [Press enter]

Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]

Your identification has been saved in
/home/gabriel/.ssh/id_rsa.
Your public key has been saved in
/home/gabriel/.ssh/id_rsa.pub.
The key fingerprint is:
01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db
Seu_email@exemplo.edu
```

Copie agora a chave pública, Math input error, para o arquivo Math input error de modo a permitir o acesso a essa máquina.

```
maquina1:~$ cd ~/.ssh
maquina1:~/.ssh$ cat id_rsa.pub >> authorized_keys
```

Na máquina 2, deve ser criado também um diretório Math input error, novamente com o comando:

```
maquina2:~$ mkdir ~/.ssh
```

Agora, você deve enviar a chave privada, Math input error, e a chave pública Math input error, da máquina 1 para máquina 2, usando o comando *scp* para isso:

```
maquina1:~$ scp ~/.ssh/id_rsa gabriel@maquina2:~/.ssh/
maquina1:~$ scp ~/.ssh/id_rsa.pub gabriel@maquina2:~/.ssh/
```

Agora, copie o arquivo Math input error para o arquivo Math input error, de modo que a máquina 1 possa fazer acesso à máquina 2 sem uso de senha:

```
maquina2:~$ cd ~/.ssh
maquina2:~/.ssh$ cat id_rsa.pub >> authorized_keys
```

Executando um programa

Para executar o programa, digamos, com 4 processos, devemos copiar o executável para o seu diretório de trabalho e digitar:

```
$ mipexec -n 4 prog
```

O comando Math input error permite opções mais elaboradas:

```
$ mpiexec -n 1 -host paraty : -n 19 prog
```

Dispara o processo com ranque 0 na máquina "paraty" e outros 19 processos divididos entre as demais máquinas. Para executar através de múltiplas máquinas:

```
$ mpiexec --hostfile hostfile -n 4 prog
```

Onde *hostfile* é um arquivo contendo o nome ou IP das máquinas onde se deseja executar a aplicação, por exemplo:

```
paraty:2
charitas:3
barra:2
```

Onde o número ao lado do nome de cada máquina indica o total de processos a serem executados. Os comentários neste arquivo são precedidos com o caractere "#". Para saber mais opções, digite:

```
$ mpiexec --help
```

Gerenciador de Recursos

Gerenciadores de recursos como SGE, PBS, SLURM são comuns em muitos *clusters* gerenciados. Nesse caso, o MPI detecta a sua existência e interage automaticamente com eles. No caso do PBS, você pode criar um arquivo *script* como a seguir:

```
# declarar um nome para esta tarefa como sample_job
#PBS -N my_parallel_job
# solicita que a tarefa seja executada na fila com nome parallel
#PBS -q parallel
# solicita um total de 4 processadores para esta tarefa (2 nós e 2 processadores por nó)
#PBS -l nodes=2:ppn=2
# solicita 4 horas de tempo de processador
#PBS -l cput=0:04:00
# combina a saída padrão e a saída de erro do PBS
#PBS -j oe
# envia um email quando a tarefa inicia e quando ela termina e aborta
#PBS -m bea
# especifica o seu endereço de e-mail
#PBS -M John.Smith@dartmouth.edu
#muda para o diretório onde a sua tarefa vai ser submetida
cd $PBS_O_WORKDIR
#inclui o caminho completo com o nome do programa MPI a ser executado
mpirun -machinefile $PBS_NODEFILE -np 4 /path_to_executable/program_name
exit 0
```

As tarefas podem ser submetidas como:

```
$ qsub -l nodes=2:ppn=2 teste.sub
```

O **mpiexec** vai saber automaticamente da existência do PBS no sistema e pergunta o número total de processadores alocados (4 no caso), e quais nós foram alocados para o processamento da tarefa. O uso é similar para outros gerenciadores de recurso. Mais opções para o comando **qsub** podem ser encontradas em https://www.jlab.org/hpc/PBS/qsub.html.

8.2 Preparação do Ambiente de Execução OpenMP

8.2.1 Compiladores

Para a utilização de um programa com OpenMP há várias opções de linguagem e compiladores disponíveis. O OpenMP pode ser usado com Fortran, C e C++, para quais destacamos os compiladores: GCC, PGI, Intel e LLVM.

Compilador GCC

O compilador **gcc** pode ser utilizado nos sistemas operacionais como Linux, MacOSX, Windows e nas diversas versões de BSD. O suporte para o OpenMP no compilador gcc teve o seguinte histórico de acordo com as suas versões:

Suporte OpenMP nas versões do GCC

Versão	**Suporte**	**Linguagens**
GCC 4.2.0	OpenMP 2.5	C/C++/Fortran
GCC 4.4.0	OpenMP 3.0	C/C++/Fortran
GCC 4.7.0	OpenMP 3.1	C/C++/Fortran
GCC 4.9.1	OpenMP 4.0	C/C++/Fortran
GCC 6.1	OpenMP 4.5	C/C++
GCC 7.1	OpenMP 4.5	Fortran (parcial)
GCC 9.1	OpenMP 5.0	C/C++ (parcial)

O comando para a compilação do programa com o compilador gcc é:

```
$ gcc omp_hello.c -fopenmp -o teste
```

É necessário definir a quantidade de *threads* que serão utilizadas antes da execução de um programa. Isso pode ser feito com uso da

variável de ambiente OMP_NUM_THREADS, cujo valor é definido com o seguinte comando:

```
$ export OMP_NUM_THREADS=4
```

ou

```
$ setenv OMP_NUM_THREADS 4
```

A execução é bem simples, sem necessidade de comando adicionais, para qualquer uma das versões de compilador:

```
$ ./teste
```

Compilador PGI

O compilador PGI oferece suporte quase completo para o OpenMP 4.5 nas linguagens C/C++/Fortran no sistema operacional Linux/x86-64 e Linux/OpenPOWER. O *offloading* é implementado para execução no hospedeiro com vários núcleos como padrão, assim como a região **parallel** é paralelizada para todas as *threads* OpenMP. As limitações conhecidas são as seguintes: as diretivas **simd** e **declare simd** não tem efeito na geração de código. O suporte para o OpenMP 3.1 é completo para as linguagens C/C++/Fortran nos sistemas operacionais MacOS e Linux.

```
$ pgcc omp_hello.c -mp -o teste
```

Compilador Intel

O compilador da Intel tem suporte para as linguagens C/C++/Fortran nos sistemas operacionais Windows, Linux, and MacOSX.

Suporte OpenMP nas versões do compilador Intel

Versão	Suporte	Linguagens

Versão	Suporte	Linguagens
Intel Compiler 12.0,13.0,14.0	OpenMP 3.1	C/C++/Fortran
Intel Compiler 15.0,16.0	OpenMP 4.0	C/C++/Fortran
Intel Compiler 17.0,18.0,19.0	OpenMP 4.5	C/C++/Fortran
Intel Compiler 19.1	OpenMP 5.0	C/C++/Fortran (parcial)
Intel Compiler 2021.1.2	GPU offloading	C/C++/Fortran

```
$ icc omp_hello.c -qopenmp -o teste
$ icc omp_hello.c -openmp -o teste
```

Para a geração de um relatório com diagnóstico de paralelização, compile o programa com os comandos a seguir:

```
$ icc -o myprog myprog.c -openmp -openmp_report2
```

Há suporte para todas as facilidades que não requerem *offloading* da versão 4.5 do OpenMP. O suporte para as construções que resultam em *offloading* que executem no hospedeiro estão disponíveis na versão 7.0. O suporte para *offloading* para dispositivos GPU está disponível na versão 8.0. O suporte para o padrão OpenMP 5.0 está em pleno desenvolvimento.

Compilador Clang

O compilador Clang do LLVM é um compilador de código aberto, que tem suporte completo para o OpenMP 4.5. O compilador Clang suporta *offloading* para arquiteturas x86_64, ARM de 64 bits e PowerPC de 64 bits e tem suporte básico para dispositivos Cuda.

Ele pode ser instalado a partir dos fontes disponíveis em http://llvm.org/releases/download.html ou com uso de pacotes pré-compilados nos sistemas operacionais disponíveis nos seus

repositórios. Para seu correto funcionamento com o OpenMP o pacote de desenvolvimento deve ser instalado junto do compilador e das bibliotecas de execução.

Fedora, CentOS e similares

Para instalação do Clang diretamente a partir repositório:

```
$ sudo dnf install clang
```

Ubuntu, Debian e similares

Para instalação do Clang diretamente do repositório:

```
$ sudo apt-get install clang
```

O Clang foi projetado para oferecer suporte a linguagens de programação C e C++ e sua execução é simples:

```
$ clang omp_hello.c -fopenmp -o teste
$ clang++ omp_hello.c -fopenmp -o teste
```

Como informação adicional, o GCC e o Clang usam bibliotecas de execução OpenMP diferentes: libgomp e libomp respectivamente.

O engenho de execução do Clang é o LLVM OpenMP, baseado no engenho de execução Intel OpenMP (código aberto) e está disponível em https://www.openmprtl.org/.

8.2.2 Variáveis de ambiente

Compilador PGI

O compilador PGI utiliza-se das variáveis de ambiente conforme definido no padrão OpenMP (veja a Seção 6.6). Atualmente a

variável de ambiente OMP_DYNAMIC não produz nenhum efeito. O valor padrão para a variável OMP_THREAD_LIMIT é 64.

Compilador GCC

- **GOMP_CPU_AFFINITY**

  Fixa *threads* a processadores específicos. A variável deve conter uma lista de processadores separados por espaço ou vírgula. Essa lista pode conter diferentes tipos de entradas: números de processadores iniciando em '0', uma faixa de processadores (M-N) ou uma faixa de processadores com um salto (M-N:S). Se ambas as variáveis de ambiente GOMP_CPU_AFFINITY e OMP_PROC_BIND estiverem definidas, então OMP_PROC_BIND tem maior precedência.

```
$ export GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"
$ export GOMP_CPU_AFFINITY=0
```

- **GOMP_DEBUG**

  Habilita a saída de depuração. A variável deve ser definida como '0' (desabilitado, o valor padrão) ou '1'.

- **GOMP_STACKSIZE**

  Define o tamanho padrão da pilha das *threads*. Tem a mesma sintaxe e comportamento da variável de ambiente OMP_STACKSIZE (Veja a Seção 6.6).

8.3 Preparação do ambiente de execução OpenACC

8.3.1 Introdução

Antes de compilar qualquer código, é importante saber quais dispositivos aceleradores estão configurados para uso no sistema. Existem alguns comandos que fornecem informações de modelos e características desses dispositivos.

Para aceleradores da NVIDIA existes os comandos *nvidia-smi* e *nvidia-settings*, que fornecem de informações de configuração como: modelo, cpus, cuda core, memória. Com o compilador da PGI também existe um comando chamado *pgaccelinfo* que fornece as principais características dos aceleradores instalados no sistema.

```
$ nvidia-smi -q | grep "Product Name"
    Product Name                    : Quadro K420
    Product Name                    : Tesla K80
    Product Name                    : Tesla K80

$ pgaccelinfo | grep "Device Name"
    Device Name:                  Tesla K80
    Device Name:                  Tesla K80
    Device Name:                  Quadro K420
```

Os aceleradores da NVIDIA necessitam que sejam instalados os *drivers* NVIDIA específicos para cada tipo de arquitetura e modelo. Conforme o modelo do acelerador, o sistema identifica um dispositivo específico e sua capacidade de computação conforme a relação a seguir.

Capacidade computacional

Arquitetura	**Dispositivo**	**capacidade de computação**
Ampere	cc80	8.0
Turing	cc75	7.5
Volta	cc72	7.2
Volta	cc70	7.0
Pascal	cc62	6.2
Pascal	cc60	6.0
Maxwell	cc50	5.0
Kepler	cc35	3.5

Arquitetura	Dispositivo	capacidade de computação
Kepler	cc30	3.0

Quando o código é compilado, ele identifica qual o *driver* NVIDIA está instalado e gera o binário do código. O comando *nvidia-smi* fornece informação da versão do *driver* NVIDIA instalado no sistema. O comando *pgaccelinfo* da PGI fornece as informações do dispositivo de GPU instalado.

```
$ nvidia-smi -q | grep "Driver Version"
Driver Version                      : 440.64.00

$ pgaccelinfo | grep "Default Target:"
Default Target:                 cc35
```

8.3.2 Compiladores

Para compilar os códigos feitos em OpenACC, gerando código para execução em GPUs, é necessário o uso de compiladores que suportem OpenACC. A PGI (Portland Group) tem uma versão disponibilizada para uso público sem a necessidade de licença. Também existe uma versão com uso de licença que permite ter acesso à equipe de suporte da PGI.

A Cray também tem seu compilador o CCE 8.6.5, para usá-lo é necessário a aquisição de uma licença, sendo que não existe um licença para uso público.

O compilador GCC também possui suporte para o OpenACC conforme a seguir:

- GCC 8: Suporta o OpenACC versão 2.0a
- GCC 9: Suporta o OpenACC versão 2.5
- GCC 10: Suporta o OpenACC versão 2.6

Os dispositivos suportados pelo GCC, no momento da escrita deste livro, eram aqueles utilizando Nvidia PTX (nvptx) e AMD GCN (GPUs

Fiji e Vega GPUs).

Nos exemplos apresentados neste livro foi utilizado o compilador da PGI disponibilizado para a comunidade, que pode ser baixado gratuitamente da página da NVIDIA[1].

Compilador PGI

Para compilar códigos em C usaremos o comando *pgcc*, e para compilação de códigos em C++ usar o comando *pgc++*. Alguns parâmetros básicos devem ser usados durante a execução dos comandos *pgcc* e *pgc++*. Esses parâmetros definem em qual dispositivo o código é executado de acordo com a arquitetura. Relação dos principais parâmetros usados no comando *pgcc*:

Parâmetros de Compilação pgcc

Parâmetro	**Descrição**
-fast	faz a otimização do código
-acc	habilita o uso de diretivas OpenACC
-Minfo=accel	informações sobre quais partes do código foram aceleradas
-Minfo=opt	informações sobre todas as otimizações de código
-Minfo=all	informações sobre todas as saídas de código
-ta=host	compila o código em modo serial
-ta=multicore	compila o código usando *threads* em CPU
-ta=nvidia	compila o código usando NVIDIA

Alguns exemplos de uso do compilador PGI e seus parâmetros básicos.

```
$ pgcc -acc -ta=nvidia -Minfo=accel main.c
$ pgc++ -acc -ta=nvidia -Minfo=accel main.cpp
```

Compilador GCC

Como dito, uma alternativa é o compilador GNU GCC que, sendo um compilador *opensource*, não requer licença para uso. Para habilitar a compilação para o OpenACC no GNU GCC, basta especificar o parâmetro *-fopenacc*.

```
$ gcc -fopenacc main.c -o teste
```

Se o programa necessitar de funções da biblioteca matemática dentro do código enviado para o acelerador, use o comando:

```
$ gcc -fopenacc -foffload=-lm main.c -o teste
```

Para sua execução basta apenas invocar o nome do programa executável na linha de comando. Ele será executado pelo processador hospedeiro e as partes que devem ser executadas pelo acelerador serão automaticamente enviadas, sem necessidade de nenhuma configuração adicional.

8.3.3 Variáveis de ambiente

Compilador PGI

Para o compilador PGI as seguintes variáveis de ambiente podem ser utilizadas:

- **PGI_ACC_TIME**

  Se a variável de ambiente PGI_ACC_TIME for definida como 1, será impresso um relatório com o tempo gasto para a movimentação de dados entre o hospedeiro e o acelerador, além do tempo de computação no acelerador. No Linux, pode ser necessário definir a variável de ambiente LD_LIBRARY_PATH para incluir o diretório onde está a biblioteca compartilhada *libnvrtc.so*, para implementar o recurso de criação de perfil.

Esse diretório varia de acordo com a versão do compilador, sendo necessário consultar a documentação para determinar o caminho correto.

- **PGI_ACC_DEBUG**

  Se for definido como 1, serve para instruir o ambiente de execução do PGI para gerar informações sobre a alocação de memória do dispositivo, movimentação de dados, inicialização do *kernel* e muito mais. PGI_ACC_DEBUG é utilizado principalmente para uso na depuração da sua execução, mas pode ser útil para entender como o programa interage com o acelerador.

- **ACC_DEVICE_TYPE**

  A variável ACC_DEVICE_TYPE tem o valor padrão *acc_device_nvidia* no compilador da PGI, exatamente como a opção do compilador -acc tem como destino *-ta=tesla* como arquitetura padrão. Os tipos de dispositivo *acc_device_default* e *acc_device_not_host* se comportam da mesma forma que *acc_device_nvidia*. O tipo de dispositivo pode ser alterado usando a variável de ambiente ou por uma chamada para a função **acc_set_device_type()** (veja a Seção 7.7).

- **ACC_DEVICE_NUM**

  No compilador da PGI, a variável ACC_DEVICE_NUM tem o valor padrão ‘0’ para o dispositivo de tipo *acc_device_nvidia*, o que é consistente com o sistema de numeração de dispositivos CUDA. Para obter mais informações, consulte a saída do comando *pgaccelinfo*. O número do dispositivo pode ser alterado usando esta variável de ambiente ou por uma chamada para a função **acc_set_device_num()**.

- **PGI_ACC_DEVICE_TYPE**

Define o tipo de dispositivo padrão a ser usado. A variável de ambiente PGI_ACC_DEVICE_TYPE se sobrepõe à variável ACC_DEVICE_TYPE. Ela controla qual dispositivo acelerador deve ser usado ao se executar regiões aceleradoras, se o programa foi compilado para usar mais de um tipo diferente de dispositivo. O valor desta variável de ambiente é definido pela implementação e, atualmente, pode ser um dos seguintes valores: NVIDIA, TESLA ou HOST, em caixa alta mesmo.

O tipo de dispositivo pode ser alterado usando essa variável de ambiente ou ainda por uma chamada para a função **acc_set_device_type (devicetype)**, onde *devicetype* no compilador PGI pode ter um dos seguintes valores:

Valores para devicetype no PGI

acc_device_none = 0	acc_device_default = 1
acc_device_host = 2	acc_device_not_host = 3
acc_device_nvidia = 4	acc_device_pgi_opencl = 7
acc_device_nvidia_opencl = 8	acc_device_opencl = 9
acc_device_current = 10	

- **PGI_ACC_DEVICE_NUM**

  Essa variável controla o número do dispositivo padrão a ser usado para executar as regiões enviadas para o acelerador. O valor desta variável de ambiente deve ser um número inteiro não negativo entre zero e o número de dispositivos conectados ao hospedeiro.

  A variável de ambiente PGI_ACC_DEVICE_NUM se sobrepõe à variável ACC_DEVICE_NUM.

- **PGI_ACC_CUDA_GANGLIMIT**

  Define o número máximo de gangues (blocos de *thread* CUDA) que serão iniciados para um *kernel*.

Compilador GCC

Para o compilador GCC são consideradas as seguintes variáveis de ambiente:

- **ACC_DEVICE_TYPE**

  Os valores que podem ser definidos são os seguintes: **nvidia, gcn e host**. O primeiro indica o uso de dispositivos aceleradores Nvidia PTX (nvptx), o segundo é utilizado para os aceleradores da AMD com arquitetura de codinome GCN (*Graphics Core Next* - GPUs Fiji e Vega), e o terceiro significa que a execução será feita por uma única *thread* no hospedeiro.

  O tipo de dispositivo pode ser alterado usando essa variável de ambiente ou por uma chamada para a função **acc_set_device_type (devicetype)**, onde *devicetype* no compilador GCC pode ter um dos seguintes valores:

  Valores para devicetype no GCC

acc_device_current = -1	acc_device_none = 0
acc_device_default = 1	acc_device_host = 2
acc_device_not_host = 4	acc_device_nvidia = 5
acc_device_radeon = 8	

- **ACC_DEVICE_NUM**

  Esse valor pode ser definido para um número inteiro não negativo, designando um dispositivo acelerador específico para ser utilizado (se houve mais de um instalado). Será emitido um erro de execução de o número especificado não corresponder a nenhum acelerador instalado.

- **GOMP_OPENACC_DIM**

Sobrescreve as dimensões padrão para computação das regiões enviadas para o acelerador. Os valores padrões são definidos em tempo de execução depois de examinar os recursos de *hardware* disponíveis. O valor ótimo para uma determinada aplicação pode não ser o padrão. Exemplos:

```
$ export GOMP_OPENACC_DIM="gang:worker:vector"
$ export GOMP_OPENACC_DIM="5120:1:32"
```

- **GOMP_DEBUG**

  A definição de um valor de Math input error pode ser adicionado às variáveis de ambiente para habilitar a depuração durante a execução do programa. Serão exibidos a movimentação de dados, o lançamento de *kernels* para execução e, para os dispositivos NPTX ativos, será exibido o código enviado para o acelerador também.

---

1. https://developer.nvidia.com/hpc-sdk↩

Referências

Abbott, Steve. 2017. "Advanced OpenACC." NVIDIA Corporation.

Amdahl, Gene M. 1967. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–85. AFIPS '67 (Spring). New York, NY, USA: ACM. https://doi.org/10.1145/1465482.1465560.

Antonio Carrillo-Ledesma, and Ismael Herrera. 2013. "Parallel Algorithms for Computational Models of Geophysical Systems." *Geofísica Internacional* 52-3: 293–309.

Ayguadé, Eduard, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. 2008. "A Proposal for Task Parallelism in OpenMP." In *A Practical Programming Model for the Multi-Core Era*, edited by Barbara Chapman, Weiming Zheng, Guang R Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, 1–12. Berlin, Heidelberg: Springer Berlin Heidelberg.

Barnes, J, and P Hut. 1986. "A hierarchical O(N log N) force-calculation algorithm." *Nature* 324: 446–49. https://doi.org/10.1038/324446a0.

Berg, Ingo. 2016. "The Barnes-Hut Galaxy Simulator." https://beltoforion.de/en/barnes-hut-galaxy-simulator/.

Bryant, Randal, and O'Hallaron David Richard. 2010. *Computer systems: a programmer's perspective*. 2nd ed. Prentice Hall.

Correa, Juliana C, and Gabriel P Silva. 2012. "Analysis and Performance Evaluation of Parallel BLAST." *I. J. Comput. Appl.* 20 (2): 112–22.

Costa, Evaldo B., Gabriel P. Silva, and Marcello Teixeira. 2018. "Avaliação de Desempenho do Montador DALIGNER em Arquiteturas Manycore." In *WSCAD 2018 - WCH ()*. São Paulo - SP, Brazil.

Flynn, Michael J. 1966. "Very high speed computers. ." *Proceedings of the IEEE* 54: 1901–9.

Gustafson, John L. 1988. "Reevaluating Amdahl's Law." *Commun. ACM* 31 (5): 532–33. https://doi.org/10.1145/42411.42415.

Herlihy, Maurice, and Nir Shavit. 2012. *The Art of Multiprocessor Programming*. Elsevier.

Hermanns, Miguel. 2002. "Parallel Programming in Fortran 95 using OpenMP." Madrid: Universidad Politecnica de Madrid. https://www.openmp.org/wp-content/uploads/F95_OpenMPv1_v2.pdf.

J, Pjesivac-Grbovic, Angskun T, Bosilca G, Fagg GE, Gabriel E, and Dongarra J. 2007. "Performance analysis of MPI collective operations." *Cluster Computing* 10 (2): 127–43. https://doi.org/10.1007/s10586-007-0012-0.

Kung, Hsiang-Tsung. 1982. "Why systolic architectures?" *IEEE Computer* 15 (01): 37–46. https://www.cs.virginia.edu/~smk9u/CS3330S20/kung_-_1982_-_why_systolic_architectures.pdf.

Marc, Snir, S Otto, Steven Huss-Lederman, D Walker, and Jack Dongarra. 1996. *MPI: the complete Reference*. Vol. 1. MIT press.

Marshall, John, Alistair Adcroft, Chris Hill, Lev Perelman, and Curt Heisey. 1997. "A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers." *JOURNAL OF GEOPHYSICAL RESEARCH-ALL SERIES*- 102: 5753–66.

MPI Forum. 2009. "MPI: A Message-Passing Interface Standard Version 2.2."

———. 2015. "MPI: A Message-Passing Interface Standard Version 3.1." MPI Forum. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

NVIDIA. 2014. "NVIDIA's Next Generation Compute Architecture: Kepler GK110/210." NVIDIA. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

OpenACC-Standard.org. 2013. "The OpenACC™ Application Programming Interface Version 2.0." OpenACC-Standard.org.

https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf.
———. 2019. *The OpenACC Application Programming Interface*. 3.0 ed. OpenACC-Standard.org. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf.
———. 2020. "The OpenACC Application Programming Interface OpenCCVersion 3.1." OpenACC-Standard.org. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf.
OpenMP ARB. 2018. "OpenMP Application Programming Interface Version 5.0." OpenMP ARB. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.
OpenMP Architecture Review Board. 2008. "OpenMP Application Program Interface Version 3.0." OpenMP ARB. https://www.openmp.org/wp-content/uploads/spec30.pdf.
———. 2013. "OpenMP Application Program Interface Version 4.0." OpenMP ARB. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.
———. 2015. "OpenMP 4.5 API C/C++ Syntax Reference Guide."
Pacheco, Peter S. 1996. *Parallel Programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
Panetta, Jairo, Thiago Teixeira, Paulo R P de Souza Filho, Carlos A da Cunha Filho, David Sotelo, Fernando M da Motta, Silvio Sinedino Pinheiro, et al. 2012. "Accelerating Time and Depth Seismic Migration by CPU and GPU Cooperation." *International Journal of Parallel Programming* 40 (3): 290–312. https://doi.org/10.1007/s10766-011-0185-2.
Quinn, Michael J, and Narsingh Deo. 1986. "An upper bound for the speedup of parallel best-bound branch-and-bound algorithms." *BIT Numerical Mathematics* 26 (1): 35–43. https://doi.org/10.1007/BF01939360.
Rahman, R. 2013. *Intel Xeon Phi Coprocessor Architecture and Tools The Guide for Application Developers*. Apress Open.
Rao, V N, and V Kumar. 1993. "On the efficiency of parallel backtracking." 4. Vol. 4. https://doi.org/10.1109/71.219757.

Rauber, T, and G Rünger. 2013. *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-37801-0.

Scott, Michael L. 2013. "Shared-Memory Synchronization." *Synthesis Lectures on Computer Architecture* 8 (2). https://doi.org/10.2200/S00499ED1V01Y201304CAC023.

Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. 2013. *Operating System Concepts*. 9th ed. New Jersey: Wiley.

Silva, Gabriel P. 2018. *Programação Paralela com MPI Um Curso Introdutório*. Edited by Gabriel P Silva. 1st ed. Rio de janeiro: Amazon. https://www.amazon.com/Programação-Paralela-com-MPI-Introdutório-ebook/dp/B079WNMF52/.

Stallings, William. 2009. *Operating Systems: internals and design principles*. 6th ed. Prentice Hall.

Tanenbaum, Andrew S, and Albert S Woodhull. 2006. *Operating systems: design and implementation*. 3rd ed. Prentice-Hall.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. 2005. "Optimization of Collective Communication Operations in MPICH." *Int. J. High Perform. Comput. Appl.* 19 (1): 49–66. https://doi.org/10.1177/1094342005051521.

Walker, D W. 1992. "Standards for message-passing in a distributed memory environment." Williamsburg, VA, USA: ORNL. https://www.osti.gov/servlets/purl/7104668.

Wang, Ping, Y Tony Song, Yi Chao, and Hongchun Zhang. 2005. "Parallel Computation of the Regional Ocean Modeling System." *International Journal of High Performance Computing Applications* 19 (4): 375–85. https://doi.org/10.1177/1094342005059115.

Wikipedia. 2020a. "Cray 1." https://www.wikiwand.com/en/Cray-1.

———. 2020b. "Manycore." https://www.wikiwand.com/en/Manycore_processor.

WILKINSON, BARRY, and MICHAEL ALLEN. 1998. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 1st ed. New York: Pearson.

Zanotto, Leandro, Anselmo Ferreira, Marcelo Matsumoto, Zanotto L Ferreira A., and Matsumoto M. 2012. "Arquitetura e programação

de GPU nvidia.” UNICAMP.